

AFRL-IF-RS-TR-2006-219
Final Technical Report
June 2006



SCALABILITY, ACCOUNTABILITY AND INSTANT INFORMATION ACCESS FOR NETWORK- CENTRIC WARFARE

Johns Hopkins University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. S468

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2006-219 has been reviewed and is approved for publication

APPROVED: /s/

ALAN J. AKINS
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Tech Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) JUNE 2006	2. REPORT TYPE Final	3. DATES COVERED (From - To) Jun 04 – Jan 06		
4. TITLE AND SUBTITLE SCALABILITY, ACCOUNTABILITY AND INSTANT INFORMATION ACCESS FOR NETWORK-CENTRIC WARFARE		5a. CONTRACT NUMBER FA8750-04-2-0232		
		5b. GRANT NUMBER		
		5c. PROGRAM ELEMENT NUMBER 62301E		
6. AUTHOR(S) Yair Amir		5d. PROJECT NUMBER S468		
		5e. TASK NUMBER SR		
		5f. WORK UNIT NUMBER SP		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Johns Hopkins University 3400 N. Charles St., Computer Science Department Baltimore Maryland 21218		8. PERFORMING ORGANIZATION REPORT NUMBER N/A		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFGA 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505		10. SPONSOR/MONITOR'S ACRONYM(S)		
		11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2006-219		
12. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA #06-462				
13. SUPPLEMENTARY NOTES				
14. ABSTRACT This project focused on one main problem: How to scale intrusion tolerant replication to wide area networks while considerably improving performance. During the last few years, there has been considerable progress in the design of intrusion-tolerant (Byzantine) replication systems. The state of the art before this project performed well on small scale systems that were usually confined to local area networks. The project developed the first hierarchical Byzantine replication architecture tailored to systems that span multiple wide area sites, each consisting of several replicas. The new architecture dramatically improves system performance (latency and throughput), availability, and manageability, for the price of extra hardware. Steward, a complete implementation of our architecture met and exceeded all performance goals and was able to withstand a white-box red team attack without being compromised even once. A side goal for the project was to look at the problem of malicious insider clients. Instead of compromising a system, malicious clients can just inject bad (but valid and authenticated) updates that corrupt information and propagate through the replicated system. By constructing an Accountability Graph between causally related updates, we demonstrate how enforcing accountability for client updates enables backtracking and state regeneration once corrupted data is discovered.				
15. SUBJECT TERMS Regenerative systems, cyber defense, scalable redundancy, accountability graph, wide-area Byzantine replication, intrusion tolerant distributed systems.				
16. SECURITY CLASSIFICATION OF: a. REPORT U b. ABSTRACT U c. THIS PAGE U		17. LIMITATION OF ABSTRACT UL	18. NUMBER OF PAGES 55	19a. NAME OF RESPONSIBLE PERSON Alan J. Akins
19b. TELEPHONE NUMBER (Include area code)				

Table of Contents

1. Summary	1
2. Introduction.....	2
3. Scalable Byzantine Replication	3
4. Accountability Graph – Coping with Malicious Clients.....	6
5. Concluding Remarks.....	8
Appendix A.....	9
Appendix B.....	40

1. Summary

This project focused on one main problem: How to scale Intrusion tolerant replication to wide area networks. Specifically, the minimal project goal was to achieve 3 times better latency for a wide area Byzantine replication system, while tolerating up to 5 compromised replicas anywhere in the system.

We invented the first hierarchical Byzantine replication architecture tailored to systems that span multiple wide area sites, each consisting of several replicas. The new architecture dramatically improves system performance (latency and throughput), availability, and manageability, for the price of extra hardware.

A full implementation of this architecture, named Steward, achieved and considerably exceeded the performance goal. For systems that are distributed over a network as wide as the continental US (50 millisecond network diameter), Steward demonstrates more than 3 times better latency and throughput as soon as there is one client in every wide area site, compared with the Byzantine Fault Tolerant (BFT) algorithm, which is the current state of the art.

The Steward system was subjected to a white-box red team attack, where the red team has complete knowledge of system design, access to its source code, and control of a number of replicas in each site. The system was not compromised, and performed well during all the red team attacks.

We think that this work is a first step toward closing the practicality gap between fault-tolerant replication and intrusion-tolerant replication over wide area networks.

A side goal for the project was to look at the problem of malicious insider clients. Instead of compromising the system, malicious clients can just use it to inject bad (but valid and authenticated) updates that propagate through the replicated system and corrupt the information in it. Clearly, since the updates are valid and the clients are authenticated (they are insiders), the computer system may not be able (at least not immediately) to detect these acts. Instead, we suggest enforcing accountability for client updates.

By constructing an Accountability Graph between causally related updates, it is possible to track the causal relationship between different information items in the system's state. Once bad data is discovered, we identify the client that injected it. We demonstrated how corrupted data that was injected subsequently by the client, as well as suspected data that causally depended on corrupted or suspected data, can be quickly marked. The system can then backtrack and regenerate its state based on non-corrupted and/or non-suspected data, and identify the extent of potential damage.

2. Introduction

An important component of Self Regenerative Systems is their resilience against malicious attackers that successfully compromise parts of the system. In this project we focused on scaling intrusion-tolerant (Byzantine) replication algorithms to large systems that can be practically deployed in realistic scenarios, and on providing resilience against malicious, but authorized clients that use their credentials to introduce bad information into the system.

We created the first hierarchical Byzantine fault-tolerant replication architecture suitable to systems that span multiple wide area sites. This architecture confines the effects of any malicious replica to its local site, reduces message complexity of wide area communication, and allows read-only queries to be performed locally within a site, for the price of additional hardware. We formally specified the algorithms of the new architecture, and proved their correctness analytically. We implemented our hierarchical architecture into a practical system, Steward, and demonstrated that it vastly improve performance (both throughput and latency) when compared with the current state of the art flat Byzantine fault-tolerant approach, over several network topologies. We verified our system implementation through a Red Team experiment where the attacker had full access to the algorithms, our proof of correctness and source code of our implementation, and full control (root) over several computer replicas. The system was not compromised in any of the attacks.

We identified a major security vulnerability in distributed systems, which refers to compromised clients that fall under adversarial control and use the system within their authorized access rights and authenticated channels to deliberately insert incorrect data. When dealing with malicious clients, a significant challenge is that when such a malicious insider is discovered, it is hard to quickly assess the scope of the damage, and identify corrupt and suspected updates.

In order to address this challenge, we introduced Accountability Graph, a mechanism that can assist applications in coping and recovering from such attacks. The tool provides accountability enforcement and causality tracking of updates and their dependencies. Upon detection of incorrect data (e.g. by an external intrusion detection mechanism or human assessment), the Accountability Graph quickly classifies all updates in the system as corrupted, suspected or not affected. We demonstrated the practicality and usefulness of this approach based on the requirements of three different applications: an open source software development project, a military common operation picture application, and a national emergency response system.

The remainder of this report is organized as follows. Section 3 is focused on the scalable Byzantine replication. Section 4 is focused on providing accountability to mitigate the effects of malicious insider (or compromised) clients. Section 5 concludes the report. The Appendix A contains:

- A paper detailing the architecture, algorithms and performance of the Steward system.
- A paper detailing the Accountability Graph mechanism.

3. Scalable Byzantine Replication

During the last few years, there has been considerable progress in the design of Byzantine fault-tolerant replication systems. The current state of the art protocols perform very well on small-scale systems that are usually confined to local area networks. However, current solutions employ flat architectures that introduce several limitations: Message complexity limits their ability to scale, and strong connectivity requirements limit their availability on wide area networks that usually have lower bandwidth, higher latencies, and exhibit network partitions.

As part of the SRS program we designed the first hierarchical Byzantine fault-tolerant replication architecture suitable for systems that span multiple wide area sites, each consisting of several server replicas. Our approach, assumes no trusted component in the entire system, other than a valid mechanism to pre-distribute private/public keys.

An implementation of our architecture, called Steward, uses a Byzantine fault-tolerant protocol within each site and a lightweight, benign fault-tolerant protocol among wide area sites. Each site, consisting of several potentially malicious replicas, is converted into a single logical trusted participant in the wide area fault-tolerant protocol. Servers within a site run a Byzantine agreement protocol to order operations locally, and they agree upon the content of any message leaving the site for the global protocol.

Guaranteeing a consistent agreement within a site is not enough. The protocol needs to eliminate the ability of malicious replicas to misrepresent decisions that took place in their site. To that end, messages between servers at different sites carry a threshold signature attesting that enough servers at the originating site agreed with the content of the message.

Using threshold signatures allows Steward to save the space and computation associated with sending and verifying multiple individual signatures. Moreover, it allows for a practical key management scheme where servers need to know only a single public key for each remote site and not the individual public keys of all remote servers.

The main benefits of our new hierarchical architecture are:

- It reduces the message complexity on wide area exchanges from order $O(N^2)$ (N being the total number of replicas in the system) to order $O(S^2)$ (S being the number of wide area sites), considerably increasing the system's ability to scale.
- It confines the effects of any malicious replica to its local site, enabling the use of a benign fault-tolerant algorithm over the wide area network. This improves the availability of the system over WANs that are prone to partitions, as only a majority of connected sites is needed to make progress, compared with at least $2f+1$ servers (out of $3f+1$) in flat Byzantine architectures (f being the maximal number of malicious replicas supported by the flat architectures).
- It allows read-only queries to be performed locally within a site, enabling the system to continue serving read-only requests even in sites that are partitioned away.
- It enables a practical key management scheme where public keys of specific replicas need to be known only within their own site.

These benefits come with a price. If the requirement is to protect against **any** f Byzantine servers in the system, Steward requires $3f+1$ servers in each site. However, in return, it is able to overcome up to f malicious servers in **each** site.

Steward's efficacy depends on using servers within a site which are unlikely to suffer the same vulnerabilities. Multi-version programming, where independently coded software implementations are run on each server, can yield the desired diversity. Newer techniques such as GENESIS from University of Virginia can automatically and inexpensively generate variation.

We demonstrated that the performance of Steward tolerating f malicious servers in *each site* is much better even compared with a flat Byzantine architecture that can tolerate only f malicious servers *total* in the system, when deployed over the same wide area topology. We also showed, for the first time, that even though it offers strong Byzantine tolerant guarantees, Steward exhibits performance comparable (though somewhat lower) with common benign fault-tolerant protocols on wide area networks.

We implemented the Steward system and a DARPA red team experiment has confirmed its practical survivability in the face of white-box attacks (where the red-team has complete knowledge of system design, access to its source code, and control of f replicas in each site). According to the rules of engagement, where a red-team attack succeeded only if it stopped progress or caused consistency errors, no attacks succeeded.

A complete account of the DARPA red team experiment for the Steward system can be found in "[Self-Regenerative Systems Red Team Assessment for STEWARD](#)", [Final Report by RABA Technologies](#).

In essence, the assessment included two parts. In the first part it is verified that Steward meets and exceeds the performance goals of the program. The latency reduction factor, as measured by RABA Technologies' people, varies from 3.72 (with one client in each of the 5 wide area sites in the experiment for a total of 5 clients), to 10.18 (with 2 clients in each site for a total of 10 clients), to 17.03 (with 3 clients in each site for a total of 15 clients) when no failures are introduced. When failures are introduced for Steward (but not for the BFT base line technology), the latency reduction factor is similar (3.71, 10.15, and 16.97 respectively). In each of these cases, the latency reduction factor was higher than 3, which was the SRS program stated goal. The latency reduction factor increases as the system size increases.

The second part of the assessment involved 6 different attacks, each with escalating severity attack scenarios. None of the attacks managed to compromise the consistency of the correct servers or block the progress of the system. Therefore, the Steward system (blue team) was considered the winner in all the cases. Below is a quote (with permission) from the RABA report:

"In summary, we assert that the STEWARD system met DARPA SRS objectives. We tested the protocol extensively and engaged tests that, according to the performer, tested their algorithms deeply. We attacked their hierarchical construct specifically. We killed servers – including critical servers. We observed the system responding and reacting – in every case

it restored itself to a normal operating status well within the time constraints and with full consistency of data.

The STEWARD team produced a resilient system, and we attribute the success of the program to three factors. First, JHU has a significant amount of experience in the high performance systems problem domain – this experience contributed to many excellent design trade-off decisions early in the process. Second, early in the conception of the project, the team spent a significant amount of time and effort developing a complete design specification for the system prior to writing any code. Unlike many of these academically developed systems – or systems developed anywhere for that matter – they fully specified what they were going to build before they started building it, and rigorously proved the correctness of their algorithms prior to implementing them. Third, at the point that they ultimately accepted the challenge of the upcoming Red Team (about six months prior to the test), they went back and re-wrote their system from the ground up using defensive programming techniques. Through this process, they discovered a number of issues with their earlier implementation, and enhanced the implementation against traditional code attacks (such as buffer overflows). Additionally, Dr. Amir has a very talented and dedicated group of researchers performing the development of this system."

The Steward work is thoroughly described in the following technical report, which is included in Appendix A:

“Steward: Scaling Byzantine Fault-Tolerant Systems to Wide Area Networks” Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen and D. Zage. Technical Report CNDS-2005-3, Distributed Systems and Networks lab, Johns Hopkins University, www.dsn.jhu.edu.

This technical report includes detailed description of the algorithms in Steward, including pseudo code that enables other researchers to replicate our work. The performance evaluation section of this technical report includes detailed benchmarks of the system in various wide area settings, comparing its throughput and latency to BFT, the baseline technology and current state of the art, for updates and read only queries.

A conference version of this technical report will appear in the *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, June 2006 and is also available on the above web site.

4. Accountability Graph – Coping with Malicious Clients

Many distributed services, including our own solution for scalable wide-area Byzantine-tolerant replication detailed in Section 3, use a set of servers that replicate the service and coordinate their actions to answer client requests, while maintaining the consistency of the data. The most basic operations performed by clients are querying the servers or updating data maintained by the servers. Security is a major concern for such systems that often operate over unsecured networks such as the Internet. Significant work conducted over the last several years to develop mechanisms for Byzantine replication, access control and intrusion detection, provides the support for designing secure distributed services. Specifically, the servers and their operating system are protected against intrusions, corrupted servers are tolerated by running Byzantine replication algorithms, access to resources is tightly enforced by using access control mechanisms, while client actions are monitored by intrusion detection systems.

Although such systems may seem difficult to attack, they often overlook that their weakest link is represented by clients, and the most critical asset is the data itself. Thus, very harmful attacks can come from compromised clients, targeting the data correctness: One or more compromised clients can *use* the system within their authorized parameters to create or inject incorrect inputs or updates to some servers. The (Byzantine) replication algorithms will propagate this information among all servers, corrupting the state of the system so that it will no longer reflect reality. In this respect, several observations are important. First, the Byzantine replication protocols running on the servers will replicate data already compromised, so they will not be able to address the attack. Second, these incorrect updates may not be detected immediately, impacting other clients subsequently querying the system and basing their decisions on the erroneous state. This creates a cascading effect in which further created updates are also erroneous because they are based on malicious data. Third, although intrusion detection mechanisms deployed in the system may eventually detect the compromised clients, assessing the extent of the damage and identifying the other components of the system that were affected is very challenging and is not provided by the mechanism mentioned above.

The effect of such an attack can be devastating for applications that are highly dependent on the correctness of their data. For example, in collaborative open-source software development (e.g. Linux), multiple individuals create or augment existing source code. The inherent interdependency between software packages enables a malicious update to one package to significantly impact other components of the system. It is important to identify the packages that may be affected by corrupt code injected into the system, and determine the risk and vulnerabilities associated with it. Other examples are command and control information systems, such as those used by the military or by emergency response personnel. In such systems, users update the state of the operational situation and make decisions based on it. Correctness of the data is critical, and any misleading information can result in loss of life. A malicious insider can inject authorized yet incorrect information that may mislead honest users and cause them, in turn, to make additional erroneous updates.

A major challenge faced by secure distributed systems is that when a malicious client insider is discovered, it is hard to quickly assess the scope of the damage, and identify

corrupt and suspected updates. Therefore, the system is usually not able to regenerate and recover to a clean state without the effects of these updates. Based on our experience building secure reliable systems, we make the observation that in the best case, this is considered an application-specific issue, and the system infrastructure provides no support in addressing it. Most of the time, this problem is not considered at all. One of our main goals in the SRS program was to raise awareness to this important problem and to show how the distributed infrastructure can assist the application in recovering from such attacks. Results based on the requirements of three different applications demonstrated the practicality of our solution.

In this project, we designed *Accountability Graph*, a generic mechanism that provides accountability enforcement and causality tracking of updates and their dependencies in a directed acyclic graph with periodic snapshots. Upon detection of incorrect data, the system traces the data to the corrupt update that generated it, and from that, the Accountability Graph enables the system to mark all causally dependent updates as corrupted or suspected. All subsequent updates made by the malicious client are marked as corrupt, and all other updates that recursively depend on corrupted updates are marked as suspicious. No less important, the system is assured that all unmarked updates are not affected by the discovered incorrect data. Our solution can use any intrusion detection mechanism (or human input) that will provide the initial detection. One or several servers forming the underlying distributed service can decide to maintain the graph, the coordination between the servers, including the ordering of the updates, will ensure that the graph looks the same at each server. There is no central authority or point of failure, any server can decide at any time if it will build the graph for events happening in the system.

We demonstrated the usefulness of our solution in three different applications: an open-source software development project, a military common operation picture application, and a national emergency response system. We showed that the overhead associated with our solution is reasonable in these cases.

The Accountability Graph work is described in the following technical report, which is included in Appendix B:

“Enhancing Distributed Systems with Mechanisms to Cope with Malicious Clients” Y. Amir, C. Danilov, J. Lane, M. Miskin-Amir and C. Nita-Rotaru. Technical Report CNDS-2005-4, Distributed Systems and Networks lab, Johns Hopkins University, www.dsn.jhu.edu.

5. Concluding Remarks

This SRS project, “Scalability, Accountability and Instant Information Access for Network-Centric Warfare”, was an intense, 18 month project conducted by the Distributed Systems and Networks lab at Johns Hopkins, with a subcontract to the Dependable and Secure Distributed Systems lab at Purdue University.

The Johns Hopkins group included Dr. Yair Amir (Project PI), Dr. Claudiu Danilov, John Lane, Jonathan Kirsch, Dr. Danny Dolev and Dr. Jonathan Shapiro.

The Purdue group included Dr. Cristina Nita-Rotaru (Subcontract PI), Josh Olsen, and David Zage.

Our research resulted in the first hierarchical Intrusion tolerant (Byzantine) replication architecture, Steward, suitable for systems that span multiple wide area sites, each consisting of several server replicas. Steward assumes no trusted component in the entire system, other than a valid mechanism to pre-distribute private/public keys. The new architecture met and exceeded the SRS program goals and sustained all the scenarios tested in a white-box red team experiment.

We also devised an Accountability Graph mechanism that can help mitigate the effects of malicious clients injecting bad (but valid and authenticated) updates to the system.

The research also introduced several new open questions.

- **Performance under attack:** The current common metrics for the evaluation of intrusion tolerant replication focus on performance in the benign case and correctness and liveness under attack. There are currently no good evaluation metrics for the system performance under attacks (perhaps because it is not well understood how to build systems that will be able to perform well under attacks). We think that future work should focus on how to construct system that will be able to perform well under attack, with the ratio of the performance under attack to performance with no attack as the proposed metric.
- **Generic hierarchical architecture:** The steward system realizes the first hierarchical Byzantine replication architecture. We believe that the architecture can be generalized such that different protocols can be plugged in for the wide area protocol and the local area protocol. If this can be accomplished, it will lead to very interesting tradeoffs between performance and guarantees. For example, we will then be able to protect against a complete site compromise by running a BFT-like algorithm between the sites. Interestingly, this will also reduce the number of replicas in each site from 16 to 10, still guarantying that no 5 malicious replicas anywhere can compromise system (although they may compromise one site).

We think that this research is a first step toward closing the practicality gap between fault-tolerant replication and intrusion-tolerant replication over wide area networks. Work along this line can become useful to C3I systems, most of them geographically spread over wide area networks, once these systems will require intrusion tolerance.

Appendix A

- “Steward: Scaling Byzantine Fault-Tolerant Systems to Wide Area Networks” Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen and D. Zage. Technical Report CNDS-2005-3, Distributed Systems and Networks lab, Johns Hopkins University, www.dsn.jhu.edu. A conference version of this technical report will appear in the *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, June 2006 and is also available on the above web site.

Steward: Scaling Byzantine Fault-Tolerant Systems to Wide Area Networks

Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, David Zage

Technical Report CNDs-2005-3 - December 2005
<http://www.dsn.jhu.edu>

Abstract—This paper presents the first hierarchical Byzantine tolerant replication architecture suitable to systems that span multiple wide area sites. The architecture confines the effects of any malicious replica to its local site, reduces message complexity of wide area communication, and allows read-only queries to be performed locally within a site for the price of additional hardware. A prototype implementation is evaluated over several network topologies and is compared with a flat Byzantine tolerant approach.

I. INTRODUCTION

During the last few years, there has been considerable progress in the design of Byzantine tolerant replication systems. The current state of the art protocols perform very well on small-scale systems that are usually confined to local area networks. However, current solutions employ flat architectures that introduce several limitations: Message complexity limits their ability to scale, and strong connectivity requirements limit their availability on wide area networks that usually have lower bandwidth, higher latencies, and exhibit network partitions.

This paper presents Steward, the first hierarchical Byzantine tolerant replication architecture suitable for systems that span multiple wide area sites, each consisting of several server replicas. Steward assumes no trusted component in the entire system, other than a valid mechanism to pre-distribute private/public keys.

Steward uses a Byzantine tolerant protocol within each site and a lightweight, benign fault tolerant protocol among wide area sites. Each site, consisting of several potentially malicious replicas, is converted into a single logical trusted participant in the wide area fault-tolerant protocol. Servers within a site run a Byzantine agreement protocol to order operations locally, and agree upon the content of any message leaving the site for the global protocol.

Guaranteeing a consistent agreement within a site is not enough. The protocol needs to eliminate the ability of malicious replicas to misrepresent decisions that took place in their site. To that end, messages between servers at different sites carry a threshold signature attesting that enough servers at the originating site agreed with the content of the message. Using threshold signatures allows Steward to save the space and computation associated with sending and verifying multiple individual signatures. Moreover, it allows for a practical key

management scheme where servers need to know only a single public key for each remote site and not the individual public keys of all remote servers.

The main benefits of our architecture are:

- 1) It reduces the message complexity on wide area exchanges from N^2 (N being the total number of replicas in the system) to S^2 (S being the number of wide area sites), considerably increasing the system's ability to scale.
- 2) It confines the effects of any malicious replica to its local site, enabling the use of a benign fault-tolerant algorithm over the wide area network. This improves the availability of the system over wide area networks that are prone to partitions, as only a majority of connected sites is needed to make progress, compared with at least $2f + 1$ servers (out of $3f + 1$) in flat Byzantine architectures.
- 3) It allows read-only queries to be performed locally within a site, enabling the system to continue serving read-only requests even in sites that are partitioned.
- 4) It enables a practical key management scheme where public keys of specific replicas need to be known only within their own site.

These benefits come with a price. If the requirement is to protect against **any** f Byzantine servers in the system, Steward requires $3f + 1$ servers in each site. However, in return, it is able to overcome up to f malicious servers in **each** site.

Steward further optimizes the above approach based on the observation that not all messages associated with the wide area fault-tolerant protocol require a complete Byzantine ordering agreement in the local site. A considerable amount of these wide area messages require a much lighter local site step, reducing the communication and computation cost on the critical path.

The paper demonstrates that the performance of Steward with $3f + 1$ servers in *each site* is much better even compared with a flat Byzantine architecture with a smaller system of $3f + 1$ *total* servers spread over the same wide area topology. The paper further demonstrates that Steward exhibits performance comparable (though somewhat lower) with common benign fault-tolerant protocols on wide area networks.

The Steward system is completely implemented and is currently undergoing a DARPA red-team experiment to assess

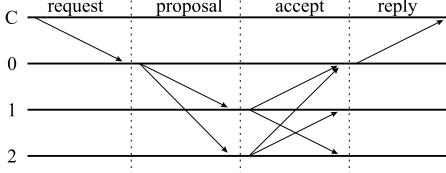


Fig. 1. Normal-case operation of the Paxos algorithm when $f = 1$. Server 0 is the current leader.

its practical survivability in the face of white-box attacks (where the red team has complete knowledge of system design, access to its source code, and control of up to f replicas in each site). We hope to be able to report on the insight gained from this activity in a final version of this paper.

The remainder of the paper is presented as follows. We provide a more detailed problem statement in Section II. We present our assumptions and the service model in Section III. We describe our protocol, Steward, and provide a sketch for a proof that it meets the specified safety and liveness properties, in Sections V and VI. We present experimental results demonstrating the improved scalability of Steward on wide area networks in Section VII. We discuss previous work in several related research areas in Section VIII. We summarize our conclusions in Section IX.

II. BACKGROUND

Our work uses concepts from fault tolerance, Byzantine fault tolerance and threshold cryptography. To facilitate the presentation of our protocol, Steward, we first provide an overview of the state-of-art work in these areas: Paxos, BFT and RSA Threshold Signatures in Sections II-A, II-B and II-C. Steward used ideas and concepts from all these algorithms.

A. Paxos Overview

Paxos [1], [2] is a well-known fault-tolerant protocol that allows a set of distributed servers, exchanging messages via asynchronous communication, to totally order client requests in the benign-fault, crash-recovery model. One server, referred to as the *leader*, has the task of coordinating the protocol. If the leader crashes or becomes unreachable, a new leader is elected. Paxos requires at least $2f + 1$ servers to tolerate f faulty servers. Since servers are not Byzantine, only one reply needs to be delivered to the client.

In the common case, in which a single leader exists and can communicate with a majority of servers, Paxos uses two asynchronous communication rounds to globally order client updates. In the first round, *Proposal*, the leader assigns a sequence number to a client update, and proposes this assignment to the rest of the servers. In the second round, *Accept*, any server receiving the proposal assents to the assigned sequence number, or *accepts* the proposal, by sending an acknowledgment to the rest of the servers. When a server receives a majority of acknowledgments – indicating that a majority of servers have accepted the proposal – the server

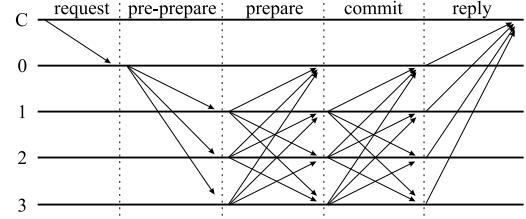


Fig. 2. Normal-case operation of the BFT algorithm when $f = 1$. Server 0 is the current leader.

orders the corresponding update. Common case operation is presented in Figure II-A.

If the leader crashes or is partitioned away, the servers run a *leader election protocol* to replace the old leader, allowing progress to resume. The leader election protocol follows a similar two-round, proposal-accept pattern, where the value proposed will be a new leader. The protocol associates a unique view number with the reign of a leader (i.e. view) and defines a one-to-one mapping between the view number and the identifier of the server acting as the leader in this view. The system proceeds through a series of views, with a view change occurring each time a new leader is elected. Proposals are thus made in the context of a given view.

Since the communication is asynchronous, multiple leaders may coexist, each issuing proposals for client requests. Paxos ensures that safety is preserved in the face of multiple leaders in two ways. First, it defines a total ordering on all proposals by attaching the view number and sequence number to each proposal. Second, with this total ordering in place, the algorithm uses an additional round of communication whenever a view change occurs to prevent conflicting requests from being ordered with the same sequence number. This round, *Prepare*, ensures that the new leader learns of any outstanding proposals that may have been ordered by a server that crashed or partitioned away. The leader collects information from a majority of servers. Since any ordered proposal was accepted by a majority of servers, and since any two majorities intersect, the ordered proposal is guaranteed to be reported to the new leader. The leader then protects a server that may have ordered the proposal (if one exists) by replaying the proposal with the same sequence number in the new view.

In summary, Paxos uses two communication rounds in the normal case (*Proposal* and *Accept*) and one additional round, *Prepare*, in addition to the leader election protocol, when a new leader is needed and a view change must take place. View changes are triggered by timeouts.

B. BFT Overview

The BFT [3] protocol addresses the problem of replication in the Byzantine model where a number of the servers can be compromised and exhibit arbitrary behavior. Similar to Paxos, BFT uses an elected leader to coordinate the protocol, and proceeds through a series of views. BFT extends Paxos into the Byzantine environment by using an additional round of communication in the common case to ensure consistency both

in and across views, and by constructing strong majorities in each round of the protocol. Specifically, BFT requires end-to-end acknowledgments from $2f + 1$ out of $3f + 1$ servers to mask the behavior of f Byzantine servers. A client must wait for $f + 1$ identical responses to be guaranteed that at least one correct server assented to the returned value.

In the common case, BFT uses three communication rounds: *Pre-Prepare*, *Prepare* and *Commit*. In the first round, the leader assigns a sequence number to a client update and proposes this assignment to the rest of the servers by multicasting a *pre-prepare* message to all servers. In the second round, a server accepts the proposed assignment by sending an acknowledgment, *prepare*, to all servers. Since a malicious leader may propose conflicting assignments, both the *pre-prepare* and *prepare* messages include the digest of the client update; this allows correct servers to differentiate acknowledgments sent in response to different *pre-prepare* messages. The first two communication rounds guarantee that correct servers agree on a total order of the updates proposed within the same view. When a server receives $2f + 1$ *prepare* messages with the same view number, sequence number, and digest as the *pre-prepare*, it begins the third round, *Commit*, by multicasting a *commit* message to all servers. A server *commits* the corresponding update when it receives $2f + 1$ matching *commit* messages. The third communication round, in combination with the view change protocol, ensures the total ordering of updates across views.

If the leader crashes, or if no progress is made, the servers initiate a view change protocol to replace the leader. View changes are triggered by timeouts. A server initiates a view change by sending a *view-change* message to all servers, suggesting a new view number (with its associated leader). When the new leader receives $2f + 1$ *view-change* messages for the same view number, it initiates a reconciliation process by sending a *new-view* message. In this process, the new leader solicits information about committed and outstanding updates from $2f + 1$ servers. Since any committed update is known to at least $f + 1$ correct servers, any set of $2f + 1$ servers will contain at least one of these correct servers; thus, the committed update will be reported to the new leader. The leader then protects servers that may have committed an update by replaying the *pre-prepare* message with the same sequence number in the new view.

In summary, BFT requires three communication rounds – *Pre-prepare*, *Prepare* and *Commit* – in the normal case, and two more communication rounds when a new leader is needed and a view change must take place.

C. Threshold Digital Signatures

Threshold cryptography [4] distributes trust among a group of participants to protect information (e.g. threshold secret sharing [5]) or computation (e.g. threshold digital signatures [6]). Threshold schemes define a threshold parameter, k , such that any set of at least k (out of n) participants can work together to perform a desired task (such as computing a digital signature), while any subset of fewer than k participants is

unable to do so. In this way, threshold cryptography offers a tunable degree of fault-tolerance: in the benign fault model, the system can function despite $(n-k)$ faults, and in the Byzantine fault model, an adversary must corrupt k participants to break the system. In particular, corrupting fewer than k participants yields no useful information. There is a natural connection between Byzantine fault-tolerance and threshold cryptography, since both distribute trust among participants and make assumptions about the number of honest participants required in order to guarantee correctness.

A (k, n) threshold digital signature scheme allows a set of n servers to generate a digital signature as a single logical entity despite $f = (k - 1)$ Byzantine faults. In a (k, n) threshold digital signature scheme, a private key is divided into n partial shares, each owned by a server, such that any set of k servers can pool their shares to generate a valid threshold signature, while any set of fewer than k servers is unable to do so. To sign a message m , each server uses its share to generate a partial signature on m , and sends the partial signature to a *combiner* server. The combiner combines the partial signatures into a threshold signature on m . The threshold signature is verified in the standard way, using the public key corresponding to the divided private key. Shares can be changed proactively [7], [8] without changing the public key, allowing for increased security and fault-tolerance, since an adversary must compromise k partial shares within a certain time window to break the system.

Since the participants can be malicious, it is important to be able to verify that the partial signature provided by any participant is valid – that is, it was generated with a share from the initial key split. This property, known as verifiable secret sharing [9], guarantees the robustness [10] of the threshold signature generation.

A representative example of practical threshold digital signature schemes is the RSA Shoup [6] scheme, which allows participants to generate threshold signatures based on the standard RSA[11] digital signature. The scheme defines a (k, n) RSA threshold signature scheme, and provides verifiable secret sharing. The computational overhead of verifying that the partial signatures were generated using correct shares is significant. The resulting threshold signature can be non-interactively verified using the same technique as the standard RSA signature.

In summary, generating a threshold signature requires one communication round and verifying the correctness of shares is an expensive operation that can be omitted in the optimistic case.

III. SYSTEM MODEL

Servers are implemented as deterministic state machines. All correct servers begin in the same initial state. Servers transition from one state to the next by applying updates to their state machines. We assume that the next state is completely determined by the current state and the next action to be applied.

We assume a Byzantine fault model. Servers are classified as either *correct* or *faulty*. Faulty servers may behave in an arbitrary manner. In particular, they can: exhibit two-faced behavior, fail to send messages, collude with other faulty servers, etc. We assume that correct servers do not crash.

Communication is asynchronous. Messages can be delayed, lost, or duplicated, but those messages that do arrive are not corrupted.

Servers are organized into wide area *sites*. Each site is identified by a unique identifier. Each server belongs to exactly one site. The network may partition into multiple disjoint *components*, each containing one or more sites. Components may subsequently remerge. Servers from sites in different components are unable to communicate with each other.

We assume that communication latency within a site is smaller than communication encountered in communication between sites.

Each site S_i has at least $3 * (f_i) + 1$ servers, where f_i is the maximum number of servers that may be faulty within S_i . For simplicity, we assume in what follows that all sites may have f faulty servers.

Clients are distinguished by unique identifiers. Clients send updates to servers within their local site, and receive responses from these servers. Each update is uniquely identified by a pair consisting of the identifier of the client that generated the update and a unique, monotonically increasing logical timestamp. Clients propose updates sequentially: a client may propose an update with timestamp $i + 1$ only after it has received a response for an update with timestamp i .

We employ digital signatures, and we make use of a cryptographic hash function to compute message digests. Client updates are properly authenticated and protected against modifications. We assume that all adversaries, including faulty servers are computationally bounded such that they cannot subvert these cryptographic mechanisms.

We also use (k,n) threshold signatures. Each site has a public key, while each server receives shares and the corresponding proofs that can be used to generate threshold signatures on behalf of the site. We assume the threshold signature scheme is cryptographically secure such that threshold signatures are unforgeable without knowing k or more secret shares.

IV. SERVICE SAFETY AND LIVENESS PROPERTIES

The protocol assigns global, monotonically increasing sequence numbers to updates to establish a global, total order. Below we define the safety and liveness properties of the STEWARD protocol. We say that:

- a *client proposes* an update when the client sends the update to a server in the local site.
- the update with sequence number i is the i^{th} update.
- a *server initiates* an update when, upon receiving the update from a client, the server forwards the update for global ordering.
- a *site initiates* an update when the leading site locally orders the update in the current global view (creating

a threshold signed proposal message which binds a sequence number to the update), and then a correct server from the site sends the proposal on the wide area for global ordering.

- a *server executes* an update with sequence i when it applies the update to its state machine. A server executes update i only after having executed all updates with a lower sequence in the global total order.
- a *site executes* an update when some correct server in the site executes the update.
- *two servers within a site are connected* if they can communicate with no communication failures.
- *two sites are connected* if every correct server of each site can communicate with every correct server of the other site with no communication failures.

DEFINITION 4.1: S1 - SAFETY: If two correct servers execute the i^{th} update, then these updates are identical.

DEFINITION 4.2: S2 - VALIDITY: Only an update that was proposed by a client (and subsequently initiated by a server) may be executed.

DEFINITION 4.3: LL1- LOCAL PROGRESS: If there exists a set, within a site, consisting of at least $2f + 1$ correct servers, and a time after which the correct members of this set are connected, then if a correct server in the set initiates an update, the site will eventually initiate the update.

DEFINITION 4.4: GL1 - GLOBAL PROGRESS: If there exists a set consisting of a majority of sites, each meeting LL1, and a time after which all sites in the set are connected, then if a site in the set initiates an update, some site in the set eventually executes the update.

V. PROTOCOL DESCRIPTION

Steward leverages a hierarchical architecture to scale Byzantine replication to the high-latency, low-bandwidth links characteristic of wide area networks. It employs more costly Byzantine fault-tolerant protocols within a site, confining Byzantine behavior to a site and allowing a more lightweight, fault-tolerant protocol to be run among sites. This results in fewer messages and communication rounds on the wide area compared to a flat Byzantine solution. The price is the need to have enough hardware within a site to overcome f malicious servers.

A site is made to behave as a single logical participant in the wide area fault-tolerant protocol through a combination of Byzantine agreement and threshold digital signatures. The servers within a site agree upon the content of any message leaving the site, and then construct a threshold signature on the message to prevent a malicious server from misrepresenting the site. One server in each site, referred to as the *representative*, coordinates the internal agreement and threshold signing protocols within the site. The representative of one site, referred to as the *leading site*, coordinates the wide area agreement protocol. If the representative of a site acts maliciously, the servers of that site will elect a new representative. If the leading site is partitioned away, the servers in the other sites will elect a new leading site.

At a higher level, Steward uses a wide area Paxos-like algorithm to globally order updates. However, the entities participating in our protocol are not single trusted participants like in Paxos. Each site entity in our wide area protocol is composed of a set of potentially malicious servers. Steward employs several intra-site protocols as building blocks at each site, to emulate a correct Paxos participant in each of the wide area algorithm steps, based on need. For example, the leader participant in Paxos unilaterally assigns a unique sequence number to an update. Instead, Steward uses an intra-site protocol that employs a BFT-like mechanism to assign a global sequence number in agreement with the servers inside the *leading site*. The *leading site* will need to present to other sites a proof that the sequence indeed was assigned. Steward uses a different intra-site protocol to threshold-sign the Paxos proposal message demonstrating that $f + 1$ *correct* servers in the *leading site* agreed to that global sequence number. The same threshold signature intra-site protocol is used to issue Paxos-like acknowledgments in non-leader sites.

In addition, Steward uses intra-site protocols that serve for Byzantine election of the new *representative* inside each site, as well as for proposing a new *leading site*.

The intra-site protocols used by Steward are as follows:

- P1-THRESHOLD-SIGN: this protocol signs a message with a threshold signature composed of $2f + 1$ shares, within a site. After executing this protocol, every correct process has a message that was signed with a threshold signature composed of $2f + 1$ shares.
- P2-ASSIGN-SEQUENCE: this protocol assigns a sequence number to an update received within a site, in the case when the representative is not suspected, and no internal view change takes place. It is invoked at the *leading site* to assign a unique sequence number to an update such that at least $f + 1$ correct servers will agree on the sequence number.
- P3-PROPOSE-LEADER-SITE: this protocol is used to generate an agreement inside a site regarding which wide area site should be the next *leading site* in the global ordering protocol.
- P4-CONSTRUCT-COLLECTIVE-STATE: this protocol provides reconciliation during a view change and generates a message describing the current state of the site, as agreed by at least $f + 1$ *correct* servers inside the site.

The high-level inter-site protocols used by Steward are listed below. Servers in multiple sites participate in these protocols.

- P5-ASSIGN-GLOBAL-ORDER: this protocol assigns a global order to each update. It uses the ASSIGN-SEQUENCE and THRESHOLD-SIGN intra-site protocols. Note its similarity to the normal-case operation of Paxos.
- P6-LAN-VIEW-CHANGE: this protocol changes the view within a site and, therefore, the local representative. A server invokes this protocol when it suspects that the current representative may be malicious. Servers in the leading site that complete this protocol are constrained such that safety is preserved across local views. A correct

constrained server will not assign a sequence number i to update u if u' may have been locally ordered with i in a previous view.

- P7-WAN-VIEW-CHANGE: this protocol changes the global view and, therefore, the leading site. A server invokes this protocol when it suspects that it is not connected to the leading site. Servers that complete this protocol are constrained such that safety is preserved across global views.

Below we provide a short description of the common case of operation of Steward, the view changes algorithms, the timers used by our protocols, and the inter-dependency between the global protocol and intra-site timeouts.

A. Data Structures and Message Types

Each server maintains several variables and data structures listed in Figure 4.

Each server can compute the *Aru* based on the corresponding history. For example, *Local_update_aru* can be computed based on the *Local_History*.

An entry in the *Pending_Proposals* is erased when it becomes less updated than the corresponding item in *Global_History*.

Each server maintains two variables *Installed_global_view* and *Installed_local_view*. Their purpose is to indicate what is the next view to be installed when there is a view change. They are set to 0 when the global or local view change protocol is invoked and 1 when the protocol ends. If *Installed_global_view* or *Installed_local_view* are 0, then *Global_view* is the new global view to be installed, and *Local_view* is the new local view, respectively.

B. The Common Case

During the common case, global progress is made and no *leading site* or site *representative* election occurs. As described above, the protocol run among sites follows a Paxos-like communication pattern and is coordinated by the *leading site*. Each round of the protocol uses one or more intra-site intra-site protocols to generate the appropriate wide area message (*proposal* and *accept*, respectively). The common case works as follows:

- 1) A client located at some site sends an update to a server in its local site. This server forwards the update to the local *representative*.
- 2) The local *representative* forwards the update to the *representative* of the *leading site*.
- 3) The *representative* of the *leading site* initiates a Byzantine agreement protocol within the site to assign a global sequence number to the update; this assignment is encapsulated in a *proposal* message. The site then generates a threshold digital signature on the constructed proposal, and the *representative* sends the signed proposal to all other sites for global ordering.
- 4) Upon receiving a signed proposal, the *representative* of each site initiates the process of generating a site

```

Update = (client_id, ts, client_update)
Server_Update

Commit = (server_id, seq, local_view, digest, t_share)

/* Messages used by the THRESHOLD-SIGN */
Sig_Share = (server_id, data, sig_share, proof)
Corrupted_Server = (server_id, data, sig_share, proof)

/* Messages used by ASSIGN-SEQUENCE */
Pre-Prepare = (server_id, seq, local_view, Update)
Prepare = (server_id, seq, local_view, digest)

Prepare_Certificate(s, v, u)= A set containing a Pre-Prepare(server_id, s, loc_v, u) message and a list of 2f distinct Prepare(server_id(i), s, loc_v', d(u)) messages with server_id ≠ server_id(i) and loc_v == loc_v'

Local_Order_Proof(s, u)= A set containing a Pre-Prepare(server_id, s, loc_v, u) message and a list of Commit(server_id(i), s, loc_v', d(u), t_share) messages satisfying Local_Ordered(s)

Local_Ordered_Update = (site_id, seq, local_view, Update, t_sig), t_sig is computed on the digest of Update; this message is equivalent to Local_Order_Proof(s, u)

Local_New_View(local_view, union, t_sig)= A view number and a set of Prepare_Certificate and Local_Order_Proof or Local_Ordered_Update messages

Proposal = (site_id, seq, global_view, local_view, Update, t_sig)
Accept = (site_id, seq, global_view, local_view, digest, t_sig)

Global_Ordered_Update(s, u)= A set containing a Proposal(site_id, s, v, u) message and a list of Accept(site_id(i), s, v', d(u)) messages, satisfying Global_Ordered(s)

/* Messages used by LAN-VIEW-CHANGE */
New_Rep = (server_id, suggested_local_view)
New_Rep_Collection = set of New_Rep messages
Preinstall_Proof = a set of 2f+1 New_Rep and the view that the l_new_rep_set proves preinstalled

/* Messages used by the SITE-ATTEMPT-WAN-VIEW-CHANGE */
VC_Attempt = (server_id, site_id, global_view, sig_share)
Global_VC = (site_id, global_view, thresh_sig)
Attempt_Proof_Request
Global_Attempt_Proof
VC_Share

/* Messages used by the CONSTRUCT-COLLECTIVE-STATE */
Request_State = (seq, ctx, ctx.view) , where ctx can be Local or Global depending on the place the protocol is invoked
Server_State = view, aru, server_id, all ordered updates and proof of the order, and accepted proposals with seq greater than a given SEQ
Server_State_Set = a set of 2f+1 Server_State distinct messages that pass validity test that view numbers is equal to the view of the corresponding Local or Global context.

```

Fig. 3. Message Types

acknowledgment (*accept*), and then sends the acknowledgment signed with a threshold signature to the representative of all other sites.

- 5) The representative of each site forwards the incoming accept messages to all local servers. A server globally orders the update when it receives signed accepts from a majority of sites. The server at the client's local site that originally received the update sends a reply back to the client.
- 6) If the client does not receive a reply to its update within a certain amount of time, it resends the update, this time broadcasting it to all servers at its site.

All site-originated messages that are sent as part of the fault-tolerant global protocol, require threshold digital signatures so that they may be trusted by other sites.

The THRESHOLD-SIGN intra-site protocol generates a $(2f+1, 3f+1)$ threshold signature on a given message. As described

in Section II, each server is assumed to have a partial share and a proof that the share was obtained from the initial secret (i.e. private key). Upon invoking the protocol on a message to be signed, the server generates a partial signature on this message. In addition, the server constructs a verification proof that can be used to confirm that the partial signature was indeed created using a valid share. Both the partial signature and the verification proof are sent to all servers within the site.

Upon receiving $2f+1$ partial signatures on a message, a server combines the partial signatures into a threshold signature on that message. The constructed signature is then verified using the site's public key (RSA verification). If the signature verification fails, then one or more partial signatures used in the combination were invalid, in which case the verification proofs provided with the partial signatures are used to identify incorrect shares; the corresponding servers are classified as malicious. The invalid shares serve as proof of corruption and

```

int Server_id: unique id of this server within the site
int Site_id: unique id of the site this server is in
bool IAm_Representative: 1 if this server is the representative for this site
bool IAm_LedgingSite: 1 if this site is the leading site
int Representative: the id of the representative for this site
int LeadingSite: the id of the leading site
A. Global Protocol Data Structure

int Global_seq: next global order sequence number to assign
int Global_view: the current global view this server is in, initialized to 0.
bool Installed_global_view: If it is 0, then Global.view is the new view to be installed.
struct globally_proposed_item {
    Proposal_struct Proposal //Can be empty
    Accept_struct List Accept_List //Can be empty
    Global_Ordered_Update_struct Global_Ordered_Update //can be empty
}
struct globally_proposed_item Global_History[] // indexed by Global_seq
int Global_aru: the global sequence number up to which this server has globally ordered all updates.

B. Leader Proposed Intermediate Data Structure

int Local_view: local view number this server is in
bool Installed_local_view: If it is 0, then Global.view is the new one to be installed.
struct pending_proposal_item {
    Pre_Prepares_struct Pre_Prepares //can be empty
    Prepares_struct List Prepare_List //can be empty
    Prepare_Cert_struct Prepare_Certificate //can be empty
    Commit_struct List Commit_List //can be empty
    Local_Ordered_Update_struct Local_Ordered_Update //can be empty
}
struct pending_proposal_item Pending_Proposals[] //indexed by Global_seq
int Pending_proposal_aru: the global sequence number up to which this server has constructed proposals

C. Local Update Data Structure

int Local_seq: next site order sequence number to assign
int Local_view: local view number this server is in
bool Installed_local_view: 0 when the view change protocol is invoked; set to 1 when the protocol ends
struct local_update_item {
    Pre_Prepares_struct Pre_Prepares //can be empty
    Prepares_struct List Prepare_List //can be empty
    Prepare_Cert_struct Prepare_Certificate //can be empty
    Commit_struct List Commit_List //can be empty
    Local_Ordered_Update_struct Local_Ordered_Update //can be empty
}
struct local_update_item Local_History[] // indexed by Local_seq
int Local_aru: the local sequence number up to which this server has locally ordered all updates.

Context definitions:
Global_Context: [A.] (only the Global Protocol Data Structure)
Pending_Context: [B.] (only Leader Proposed Intermediate Data Structure)
Local_Context: [C.] (only the Local Update Data Structures)
Order_Context: [A.] and [B.] combined (the union of the two data structures)

D. Client Related Data Structures

struct client_record {
    Update_struct Update
    int view_num
    int seq
    bool is_ordered
}
struct client_record Client_Records[] //indexed by client_id

```

Fig. 4. Data Structures Maintained by Each Server

can be broadcast to all local servers. Further messages from the corrupted servers are ignored.

Once the *representative* of the *leading site* receives an update from a client (either local or forwarded by the *representative* of a different site), it assigns a sequence number to this update by creating a proposal message that will then be sent to all other sites. The sequence number is assigned in agreement with other correct servers inside the site, masking the Byzantine behavior of malicious servers. The ASSIGN-

SEQUENCE intra-site protocol is used for this purpose. The protocol consists of three rounds, the first two of which are similar to the corresponding rounds of the BFT protocol: the site *representative* proposes an assignment by sending a *pre-prepare* message to all servers within the site. Any server receiving the *pre-prepare* message sends to all servers a *prepare* message as acknowledgment that it accepts the representative's proposal. At the end of the second round, any server that has received 2f *prepare* messages, in addition

to the *pre-prepare*, for the same view and sequence number, invokes the THRESHOLD-SIGN intra-site protocol to generate a threshold signature on the representative’s proposal.

Upon completion of the ASSIGN-SEQUENCE protocol, the *representative* sends the proposal message for global ordering on the wide area to the representatives of all other sites.

Each site’s representative receiving the proposal message forwards it to the other servers inside the site, and invokes the THRESHOLD-SIGN protocol to generate an acknowledgment (*accept*) of the proposal. The representative of the site then sends back the threshold signed *accept* message to the representatives of all other sites. Each *representative* will forward the *accept* message locally to all servers inside their site. A server within a site globally orders the update when it receives *accept* messages from a majority of sites.

C. View Changes

The above protocol describes the common-case operation of Steward. However, several types of failure may occur during system execution, such as the corruption of one or more site representatives, or the partitioning of the leader site. Such failures require delicate handling to preserve both safety and liveness.

If the representative of a site is faulty, the correct members of the site select a new representative by running a local view change protocol, after which progress can resume. The local view change algorithm preserves safety across views, even if consecutive representatives are malicious. Similarly, the *leading site* that coordinates the global ordering between the wide area sites can be perceived as faulty if no global progress is made. In this case, a global view change occurs. View changes are triggered by timeouts, as described in Section V-E

Each server maintains a local view number and a global view number. The local view number maps to the identifier of the server’s current site representative, while the global view number maps to the identifier of the wide area leader site. The local and global view change protocols update the server’s corresponding view numbers.

We first introduce the CONSTRUCT-COLLECTIVE-STATE intra-site protocol, which is used as a building block in both the local and global view change protocols.

The CONSTRUCT-COLLECTIVE-STATE protocol generates a message describing the current state of a site, as agreed by at least $f + 1$ *correct* servers within the site. The constructed message is referred to as a *union* message. The *representative* of a site invokes the protocol by sending a sequence number to all servers inside the site. Upon receiving the invocation message, all servers send to the *representative* a message containing updates they have ordered and/or acknowledged with a higher sequence number than the *representative*’s number. The *representative* computes a union on the contents of $2f + 1$ of these messages, eliminating duplicates and using the latest update for a given sequence number if conflicts exist. The *representative* packs the contents of the union into a message and sends the message to all servers in the site. Upon receiving such a union message, each server updates its own state with

missing updates as needed, generates a partial signature on the message, and sends the signed message to all servers within the site. A server then combines $2f + 1$ such partial signatures into a single message that represents the updates that the site ordered or acknowledged above the original sequence number.

Local view change: The local view change protocol is similar to the one described in [3]. It elects a new site *representative* and guarantees that correct servers cannot be made to violate previous safety constraints.

The protocol is invoked when a server at some site observes that global progress has not been made within a timeout period, and is used at both the *leading site* and non-leader sites. A server that suspects the representative is faulty increases its local view number and sends to all local servers a *new-representative* message, which contains the proposed view number. Individual servers increase their proposed local view in a way similar to [3]. Upon receiving a set of $2f + 1$ *new-representative* messages proposing the same view number (and, implicitly, a new representative), the new representative computes the sequence number of the highest update ordered, such that all updates with lower sequence numbers were ordered. We call this sequence number “ARU” (All Received Up-to). The new representative then invokes the CONSTRUCT-COLLECTIVE-STATE protocol based on its ARU. Finally, the new representative invokes the ASSIGN-SEQUENCE protocol to replay all pending updates that it learned from the signed union message.

Global view change: In the global view change protocol, wide area sites exchange messages to elect a new *leading site* if the current one is suspected to be faulty (partitioned away or with fewer than $2f + 1$ correct servers). Each site runs an intra-site protocol, PROPOSE-LEADER-SITE, to generate a threshold-signed message containing the global view number that the site has agreed to propose.

The PROPOSE-LEADER-SITE protocol is invoked in a distributed fashion. Upon suspecting that the *leading site* is faulty, a server within a site increases its global view number and generates a partial signature on a message that proposes the new view. Upon receiving $2f + 1$ partial signatures for the same global view number, the local *representative* combines the shares to construct the site’s proposal. To ensure liveness, a server already suspects the *leading site*, and that receives $f + 1$ partial signatures referring to global view numbers higher than its own, updates its global view number to the smallest value of the $f + 1$ view numbers, and sends a corresponding partial signature to the other servers in the site.

If enough servers in a site invoke the PROPOSE-LEADER-SITE protocol, the representative of that site will issue the resultant threshold-signed *new-leading-site* message that contains the identifier of that site and the proposed global view number. When the *representative* of the new *leading site* receives a majority of such messages proposing the same global view, it starts a local reconciliation protocol by invoking the CONSTRUCT-COLLECTIVE-STATE protocol on its own ARU. We call the highest sequence of an ordered update in the resulting union message, below which all lower sequence

```

THRESHOLD_SIGN(Data_s data, int server_id):
A1. Sig_Share ← GENERATE_SIGNATURE_SHARE(data, server_id)
A2. SEND to all local servers: Sig_Share

B1. Upon receiving a set, Sig_Share_Set, of 2f+1 Sig_Share from distinct servers
B2.   signature ← COMBINE(Sig_Share_Set)
B3.   if VERIFY(signature)
B4.     return signature
B5.   else
B6.     for each S in Sig_Share_Set
B7.       if NOT VERIFY(S)
B8.         REMOVE(S, Sig_Share_Set)
B9.         ADD(S.server_id, Corrupted_Servers_List)
B10.        Corrupted_Server ← CORRUPTED(S)
B11.        SEND to all local servers: Corrupted_Server
      continue to wait for more Sig_Share messages

```

Fig. 5. THRESHOLD-SIGN Protocol

```

ASSIGN-SEQUENCE(Message update, Context ctx, int server_id):
A1. if Representative
A2.   ctx.seq++
A3.   SEND to all local servers: Pre-Prepare(update, ctx.seq, ctx.view)

B1. Upon receiving Pre-Prepare(update, seq, view)
B2.   if NOT CONFLICT(Pre-Prepare, ctx)
B3.     SEND to all local servers: Prepare(update, seq, view)

C9. Upon receiving 2f Prepare for which NOT CONFLICT(Prepare, ctx)
C10.   ordered_update ← invoke THRESHOLD_SIGN(update, server_id)

```

Fig. 6. ASSIGN-SEQUENCE Protocol

```

ASSIGN-A-GLOBAL-ORDER(Message update):
A1. if LeadingSite and Representative
A2.   Proposal ← invoke ASSIGN-SEQUENCE(update, Global_History)
A3.   SEND to all sites: Proposal

B1. Upon receiving Proposal(site_id, seq, global_view, local_view, update, t_sig)
B2.   if NOT LeadingSite
B3.     if Representative
B4.       SEND to all local servers: Proposal
B5.       if Proposal.global_view ≥ Global_view
B6.         Global_view ← Proposal.global_view
B7.         Accept ← invoke THRESHOLD_SIGN(Proposal, Server_id)
B8.         SEND to all sites: Accept
B9.     if NOT Representative
B10.    if Proposal.global_view ≥ Global_view
B11.    Global_view ← Proposal.global_view
B12.    Accept ← invoke THRESHOLD_SIGN(Proposal, Server_id)

C1. Upon receiving an Accept at any site
C3.   SEND to all local servers: Accept

D1. Upon receiving a majority of Accepts at any site
C2.   return

```

Fig. 7. ASSIGN-A-GLOBAL-ORDER Protocol

numbers are ordered, “Site ARU”. The *representative* of the new *leading site* invokes the THRESHOLD-SIGN protocol on a message containing the Site ARU, and sends the resulting threshold-signed message to the representatives of all other sites. Based on the Site ARU received, the representatives of the non-leader sites invoke the CONSTRUCT-COLLECTIVE-STATE protocol and send the resultant union message back to the representative of the new *leading site*. A set of union messages from a majority of sites is used by servers in the *leading site* to constrain the messages they will generate in the new view so that safety is preserved.

D. Updating Data Structures

E. Timeouts

Steward relies on timeouts to detect problems with the representatives in different sites or with the leading site. Our protocols do not assume synchronized clocks; however, we do assume that the rate of the clocks at different servers is reasonably close. We believe that this assumption is valid considering today’s technology. Below we provide details about the timeouts in our protocol.

Local representative (T1): This timeout expires at a server of a non-leading site to replace the representative once no (global) progress takes place for that period of time. Once the timeout expires at $f + 1$ servers, the local view change protocol takes

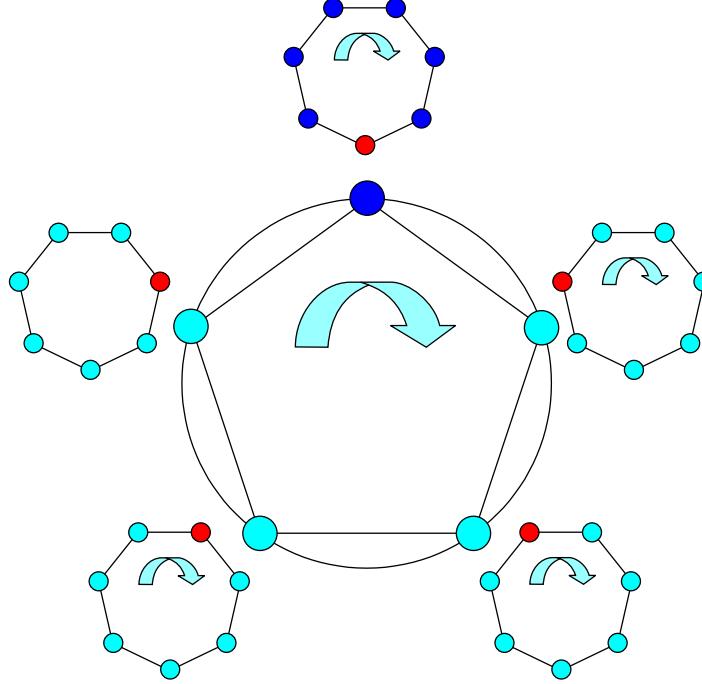


Fig. 8. Steward system having five sites with seven servers in each site. The representatives are colored red. The leader site is colored dark blue.

```

LAN-VIEW-CHANGE:
A1. Stop Timer Local_T
A2. Local_View++
A3. New_Rep ← NEW_REPRESENTATIVE(Server_id, Local_View)
A4. SEND to all local servers: New_Rep

B1. Upon receiving a set F of f+1 New_Rep with view greater than mine from distinct servers:
B2.   Local_view ← MIN_view(F)
B3.   New_Rep ← NEW_REPRESENTATIVE(Server_id, Local_view)
B4.   SEND to all local servers: New_Rep

C1. Upon receiving a set, New_Rep_Set, of 2f+1 distinct New_Rep for same view equal to mine:
C3.   if not new representative
C4.     Set Timer Local_T = L_Expiration_Time
C5.   if new representative
C6.     New_Rep_Collection ← CONSTRUCT_BUNDLE(New_Rep_Set)
C7.     SEND to all local servers: New_Rep_Collection
C8.     union ← Invoke CONSTRUCT_COLLECTIVE_STATE(Local_aru, Local_History)
C9.     Invoke ASSIGN_SEQUENCE for each unordered update

D1. Upon timeout expiration:
D2.   Local_view++
D3.   Stop Timer Local_T
D4.   L_Expiration_Time *= 2
D5.   New_Rep ← NEW_REPRESENTATIVE(Server_id, Local_view)
D6.   SEND to all local servers: New_Rep

```

Fig. 9. LAN-VIEW-CHANGE Protocol

place. T1 should be higher than 3 times the wide area network round-trip to allow a potential global view change protocol to complete without changing the local representative.

Leading site representative (T2): This timeout expires at a server at the leading site to replace the representative once no (global) progress takes place for that period of time. T2 should be large enough to allow the representative to communicate with a majority of the sites. Specifically, since not all sites may be lined up with correct representatives at the same time, T2 should be chosen such that each site can replace

its representatives until a correct one will communicate with the *leading site*; the site needs to have a chance to replace $f + 1$ representatives within the T2 time period. Thus, we need that $T2 > (f+2)*maxT1$, where $maxT1$ is an estimate of the largest T1 at any site. The $(f+2)$ covers the possibility that when the leader site elects a representative, the T1 timer is already running at other sites.

Leading site (T3): This timeout expires at a site to replace the leading site once no (global) progress takes place for that period of time. Since we choose T2 to ensure a single

```

SITE-ATTEMPT-WAN-VIEW-CHANGE:
A1. L_VC_Attempt ← GEN_VIEW_CHANGE_ATTEMPT(Global_view, Server_id)
A2. SEND to all local servers: L_VC_Attempt

B1. Upon receiving L_VC_Attempt(v) from server s
B2.   if v > Global_view + 1
B3.     Attempt_Proof_Request ← GEN_ATTEMPT_PROOF_REQUEST()
B4.     SEND to s: Attempt_Proof_Request

C1. Upon receiving an Attempt_Proof_Request from server s:
C2.   Global_Attempt_Proof ← GEN_GLOBAL_ATTEMPT_PROOF()
C4.   My_Global_Attempt_Proof ← Global_Attempt_Proof
C3.   SEND to server s: Global_Attempt_Proof

D1. Upon receiving Global_Attempt_Proof message, p, for global view v:
D2.   if Global_view < v
D3.     My_Global_Attempt_Proof ← p
D4.     Global_view ← v
D5.     L_VC_Attempt ← GEN_VIEW_CHANGE_ATTEMPT(Server_id, Global_view)
D6.     SEND to all local servers: L_VC_Attempt

E1. Upon receiving a set of 2f+1 distinct L_VC_Attempt for view greater than or equal to mine
E2.   L_VC_Share ← GENERATE_SIGNATURE_SHARE(Global_view, Server_id)
E3.   SEND to all local servers: L_VC_Share

F1. Upon receiving a set, L_VC_Share_Set, of 2f+1 distinct L_VC_Shares for Global_view
F2.   My_global_attempt_proof ← GEN_GLOBAL_ATTEMPT_PROOF(L_VC_Share_Set)
F3.   Global_VC ← COMBINE(L_VC_Share_Set)
F4.   return Global_VC

```

Fig. 10. SITE-ATTEMPT-WAN-VIEW-CHANGE Protocol

```

WAN-VIEW-CHANGE:
When the new leader site receives a majority of Global_VC messages, it constructs the Prepare(aru, view) message by invoking P4. Non-leader sites respond to this message with Prepare.OK.

A1. Upon Suspect leader site trigger:
A2.   G_Expiration_Time ← Default_Global_Timeout
A3.   Global_view ← Global_view + 1
A4.   Global_VC ← invoke SITE-ATTEMPT-WAN-VIEW-CHANGE
A5.   if Representative
A6.     Send to all sites: Global_VC

B1. Upon receiving Majority Global_VC where Global_VC.view_num = Global_view
B2.   if representative of leader site
B3.     (ConstraintMessage, AruMessage) ← Invoke CONSTRUCT-COLLECTIVE-STATE(local_aru, C)
B4.     Send to all sites: AruMessage
B5.   else
B6.     Set Timeout Global_T ← G_Expiration_Time

C1. Upon receiving Global_VC where Global_VC.view_num > Global_view
C2.   if already suspecting leader site and Global_T not set
C3.     Global_view ← Global_VC.view_num
C4.     Global_VC ← invoke SITE-ATTEMPT-WAN-VIEW-CHANGE
C5.   if representative
C6.     Send to all sites: Global_VC

D1. Upon receiving AruMessage
D2.   if AruMessage.view_num >= Global_view
D3.     Global_view ← AruMessage.view_num
D4.   Cancel Timeout Global_T if set
D5.   (CM_Pending, ...) ← Invoke CONSTRUCT-COLLECTIVE-STATE(AruMessage.aru, Pending_Proposals)
D6.   (CM_Global, ...) ← Invoke CONSTRUCT-COLLECTIVE-STATE(AruMessage.aru, Global_History)
D7.   Send to leader site: CM_Global

E1. Upon receiving ConstraintMessage
E2.   if server in leader site
E3.     if representative of leader site
E4.       Send to all local servers
E5.     if Majority of ConstraintMessage
E6.       Apply to data structures

F1. Upon expiration of Timeout Global_T:
F2.   Suspect leader site F3.   T = T*2

```

Fig. 11. WAN-VIEW-CHANGE Protocol

```

CONSTRUCT-COLLECTIVE-STATE(int seq, Context ctx):
A1. if Representative
A2. Request_State ← GEN_REQUEST_STATE(seq, ctx, ctx.Global_view) A3. SEND to all local
servers: Request_State(seq, ctx, ctx.Global_view)

B1. Upon receiving Request_State(s, c, v)
B2. if v == ctx.Global_view
B3. compute LARU = Local ARU in *Context.history
B4. if L_ARU < s
B5. Request missing ordered updates from representative
B6. if L_ARU geq s
B7. Server_State = Construct_Server_State(*Context, s)
B8. SEND to all local servers: Server_State

C1. if representative
C2. Upon receiving a set, Server_State_Set, of 2f+1 distinct Server_State messages that
pass validity test with view numbers == *Context.Global_view
C4. Collected_Servers_State ← Construct_Bundle(Server_State_Set)
C5. SEND to all local servers: Collected_Servers_State

D1. Upon receiving Collective_State with view numbers == *Context.Global_view
D2. union ← ConstructUnion(Collected_Servers_State )
D3. ConstraintMessage ← invoke THRESHOLD-SIGN(union)
D4. union_aru ← Extract_Aru(union)
D5. AruMessage ← invoke THRESHOLD-SIGN(unio_aru)
D6. Apply union to *Context.history
D7. return (ConstraintMessage, AruMessage)

```

Fig. 12. CONSTRUCT-COLLECTIVE-STATE Protocol

```

UPDATE-GLOBAL-HISTORY:
case message:
A1. Proposal(id, s, gl_v, u):
A2. if Global_History[s].Proposal is empty
A3. Global_History[s] <- Proposal(id, s, gl_v, u)
A4. if Global_History[s].Proposal contains Proposal(id', s, gl_v', u')
A5. if gl_v' ≥ gl_v
A6. ignore Proposal
A7. if gl_v' < gl_v
A8. Global_History[s] <- Proposal(id, s, gl_v, u)

B1. Accept(id, s, gl_v, d(u)):
B2. if Global_History[s].Proposal is empty
B3. ignore Accept
B4. if Global_History[s].Accept_List is empty
B5. Global_History[s].Accept_List <= Accept(site_id, s, gl_v, d(u))
B6. if Global_History[s].Accept_List contains any Accept(site_id, s, gl_v', d(u'))
B7. if gl_v > gl_v'
B8. discard all Accepts in Global_History[s]
B9. Global_History[s] <= Accept(site_id, s, gl_v, d(u))
B10. if gl_v == gl_v' and Global_History[s] does not contain Accept(site_id, *)
B11. Global_History[s] <= Accepts(site_id, s, gl_v, d(u))
B12. if gl_v < gl_v'
B13. ignore Accept
B14. if Global_Ordered_Ready(s)
B15. Construct Global_Ordered_Update from Proposal and list of Accepts
B16. Global_History.Global_Ordered_Update = Global_Ordered_Update

C1. Global_Order_Update(s, gl_v, u):
C2. if not Global_Ordered(s)
C3. Global_History[s].Global_Ordered_Update <- Global_Ordered_Update(s, gl_v, u)
C4. else
C5. ignore Global_Order_Update

```

Fig. 13. Rules of applying a message to Global History assume that there is no conflict, i.e. Conflict(message, Global_History) == FALSE

communication round with every site, and since the leading site needs at least 3 rounds to prove progress, in the worse case, the leading site must have a chance to elect 3 correct representatives to show progress, before being replaced. Thus, we need $T3 = (f + 3)T2$.

Client timer (T0): This timeout expires at the client, triggering it to inquire the status of its last update by interacting with various servers at the site. T0 can have an arbitrary value.

Timeouts management: Servers send their timers estimation ($T1, T2$) on global view change messages. The site representative disseminates the $f + 1$ st highest value (the value for which f higher or equal values exist) to prevent the faulty servers from injecting wrong estimates. Potentially, timers can be exchanged as part of local view change messages as well. The leading site representative chooses the maximum timer of all sites with which communicates to determine $T2$ (which in turn determines $T3$). Servers estimate the network round-

```

UPDATE-LOCAL-HISTORY:
case message:
A1.   Local_New_View(l_view, union, t_sig):
A2.     if l_view ≠ Local_view
A3.       ignore Local_New_View
A4.     if Installed.local_view == 0
A5.       for every s > Local_aru
A6.         discard Local_History[s].Pre-Prepare
A7.         discard all Prepare in Local_History[s].Prepare_List
A8.         discard all Commit in Local_History[s].Commit_List
A9.       Apply union to Local_History according to the rules below
A10.      Installed_local_view = 1

B1.   Pre-Prepare(server_id, seq, l_view, update):
B2.     if Local_History[s].Pre-Prepare is empty
B3.       Local_History[s].Pre-Prepare ← Pre-Prepare(server_id, seq, l_view, update)
B4.     else
B5.       ignore Pre-Prepare(server_id, seq, l_view, update)

C1.   Prepare(server_id, seq, local_view, digest):
C2.     if Local_History[s].Pre-Prepare is empty
C3.       ignore Prepare
C4.     if Local_History[s].Prepare_List contains a Prepare with server_id
C5.       ignore Prepare
C6.     else
C7.       Local_History[s].Prepare_List <= Prepare(server_id, seq, local_view, digest)
C8.       if Prepare_Certificate_Ready(s)
C9.         Construct_Prepare_Certificate(Local_History[s].Pre-Prepare, Local_History[s].Prepare_List)
C10.        Local_History[s].Prepare_Certificate ← Prepare_Certificate

D1.   Commit(server_id, seq, l_view, digest, t_share):
D2.     if Local_History[s].Pre-Prepare is empty
D3.       ignore Commit(server_id, seq, l_view, digest, t_share)
D4.     if Local_History[s].Commit_List contains a Commit with server_id
D5.       ignore Commit(server_id, seq, l_view, digest, t_share)
D6.     else
D7.       Local_History[s].Commit_List <= Commit(server_id, seq, l_view, digest, t_share)
D8.       if Local_Ordered_Ready(s)
D9.         Construct_Local_Ordered_Update(Local_History[s].Pre-Prepare, Local_History[s].Commit_List)
D10.        Local_History[s].Local_Ordered_Update ← Local_Ordered_Update

E1.   Prepare_Certificate(s, l_v, u):
E2.     if Local_History[s] contains a Prepare_Certificate(s, l_v', u')
E3.       if l_v' < l_v
E4.         Local_History[s] ← Prepare_Certificate(s, l_v, u)
E5.       else
E6.         ignore Prepare_Certificate(s, l_v, u)
E7.     else
E8.       Local_History[s] ← Prepare_Certificate(s, l_v, u)

F1.   Local_Ordered_Update(s, u):
F2.     if Local_Ordered(s)
F3.       discard Local_Ordered_Update(s, u)
F4.     else
F5.       Local_History[s].Local_Ordered_Update ← Local_Ordered_Update(s, u)

```

Fig. 14. Rules of applying a message to Local History assume that there is no conflict, i.e. $\text{Conflict}(\text{message}, \text{Local_History}) == \text{FALSE}$

trip according to various interactions they have had. They can reduce the value if communication seems to improve.

Proof: Following Lemma 6.1, Lemma 6.2 and Lemma 6.3, the Claim is proved. ■

VI. PROOF

A. Strategy

In this section we show that Steward provides the properties specified in Section IV. We prove individual properties for each of the building blocks, providing safety and liveness guarantees. The building block algorithms are listed in Section V.

B. Proof of ASSIGN-SEQUENCE Protocol

Claim: If a correct representative invokes P1 on some data (within a Context), and all correct servers are connected, then P1 returns the data with a sequence number and a signature at $f+1$ correct servers, or else a view change will take place.

OBS 1: The correct representative will assign a sequence number s to the update (data). If a view change does not take place, a correct representative will not assign the same sequence number to different updates (Lines A2-A3) in protocol P1.

OBS 2: Following the rule RL1 and lines B2-B3 in $\text{Conflict}()$ function, for any sequence number s , a correct server will only maintain L-Pre-Prepare messages sent by the current representative.

OBS 3: Following Obs 1 and Obs 2, an L-Pre-Prepare message sent by a correct representative will not conflict with any other Pre-Prepare messages

```

boolean Global_Ordered(s):
    if Global_History[s].Ordered_Update is not empty
        return TRUE
    return FALSE

boolean Global_Ordered_Ready(s):
    if Global_History.Proposal[s] contains a Proposal(site_id, s, gl_v, u)
        if Global_History[s].Accept_List contains (majority-1) of distinct
            Accept(site_id(i), s, gl_v, d(u)) with site_id(i) ≠ site_id
            return TRUE
        if Global_History[s].Accept_List contains a majority of distinct
            Accept(site_id(i), s, gl_v', d(u)) with gl_v >= gl_v'
            return TRUE
    return FALSE

boolean Local_Ordered(s, *Context):
    if Context.Local_History[s].Ordered_Update is not empty
        return TRUE
    return FALSE

boolean Local_Ordered_Ready(s, *Context):
    if Context.Local_History.Proposal[s] contains a Pre-Prepare(server_id, s, loc_v, u)
        if Context.Local_History[s].Commit_List contains 2*f+1 of distinct
            Commit(server_id(i), s, loc_v, d(u), t_share') with digest(u) == d(u)
            return TRUE
    return FALSE

boolean Prepare_Certificate_Ready(s, *Context):
    if Context.Local_History.Proposal[s] contains a Pre-Prepare(server_id, s, loc_v, u)
        if Context.Local_History[s].Prepare_List contains 2*f of distinct
            Prepare(server_id(i), s, loc_v, d(u)) with server_id ≠ server_id(i) and digest(u) == d(u)
            return TRUE
    return FALSE

boolean Conflict(message, *Context):
    case message
    Pre-Prepare(server_id, seq, local_view, Update):
        if server_id ≠ local_view mod num_servers_in_site
            return TRUE
        if local_view ≠ Context.Local_view
            return TRUE
        if Context.Local_History[s].Pre-Prepare(server_id, seq, l_view, u') exists and u' ≠ Update
            return TRUE
        if Context.Local_History[s].Prepare_Certificate(seq, l_view', u') exists and u' ≠ Update
            return TRUE
        if Context.Local_History[s].Local_Ordered_Update(site_id, seq, l_view', u', t_sig) exists
            if u' ≠ Update or l_view' > local_view
                return TRUE

    Prepare(server_id, seq, local_view, di):
    Commit(server_id, seq, local_view, D, t_share):
        if local_view ≠ Context.Local_view
            return TRUE
        if Context.Local_History[s].Pre-Prepare(server_id', seq, local_view, u) exists and digest(u) ≠ d
            return TRUE
        if Context.Local_History[s].Local_Ordered_Update(site_id, seq, l_view', u, t_sig) exists and
            if digest(u) ≠ d or l_view' > local_view
                return TRUE

    Proposal((site_id, seq, global_view, local_view, Update, t_sig):
        if global_view ≠ Context.Global_View
            return TRUE
        if Context.Global_History[s].Global_Ordered_Update(seq, g_view', u') exists
            if u' ≠ Update or g_view' > global_view
                return TRUE

    Accept(site_id, seq, global_view, local_view, d, t_sig):
        if global_view ≠ Context.Global_View
            return TRUE
        if Context.Global_History[s].Proposal(site_id, s, g_view, l_view, u, t_sig) exists and digest(u) ≠ d
            return TRUE
        if Context.Global_History[s].Global_Ordered_Update(seq, g_view', u') exists
            if digest(u') ≠ d or g_view' > global_view
                return TRUE
    return FALSE

```

Fig. 15. Predicates

mantained at any correct servers. As a consequence, upon receiving a L-Pre-Prepare message from a correct representative, Conflict() function at a correct server will only return TRUE if that server has a conflicting Prepare-Certificate or a conflicting Local_Ordered messsage.

OBS 4: According to Obs3 and Lines B1-B4 in P1, any correct server, upon receiving a L-Pre-Prepare message from a correct representative, will send a L-Prepare message unless it has a conflicting Prepare-Certificate or a conflicting Local_Ordered message.

Lemma 6.1: If a correct representative sends a L-Pre-Prepare(d, s, v) and no view change takes place, then at least $f+1$ correct servers will receive its message and at least 2^*f distinct and matching L_Prepares(d, s, v) messages from servers other than the representative.

Proof: Since the correct servers are connected, all the correct servers will receive the L-Pre-Prepare message sent by the correct representative.

If no correct server has a conflicting Prepare-Certificate or a conflicting Local_Ordered update to the L-Pre-Prepare(d, s, v) message, then all correct servers other than the representative will send a Prepare(d, s, v) message (Obs 4). This implies that all correct servers will receive at least 2^*f Prepare(d, s, v) messages in addition to the L-Pre-Prepare message.

Some correct servers may have conflicting Prepare-Certificates or Local_ordered_Updates. These servers will not send a Prepare message. If fewer than $f+1$ correct servers receive 2^*f Prepare messages then less than 2^*f+1 total servers will invoke P2. (Line C1-C2 in protocol P1) Following Property 2 of P2, P1 will not be able to complete, so no progress will be made, and this results in a view change. ■

Lemma 6.2: If at least $f+1$ correct servers receive a L-Pre-Prepare(d, s, v) and at least 2^*f distinct and matching L_Prepares(d, s, v) messages from servers other than the representative then, if P1 returns at any correct server and no view change takes place, the data and sequence number returned at that server will be the data d and sequence number s in the L-Pre-Prepare message.

Proof: If at least $f+1$ correct serves receives a L-Pre-Prepare(d, s, v) and at least 2^*f distinct and matching L_Prepares(d, s, v) messages from servers other than the representative, then at least $f+1$ correct servers will invoke P2 with (d, s, v) in the current view. Therefore, there cannot be a set of 2^*f+1 servers invoking P2 on any (d', s) in the current view.

Consequently, if P2 (and implicitly P1) completes at any correct server, it will only complete on (d, s). ■

Lemma 6.3: If a correct representative invokes P1, then either P1 completes at least $f+1$ correct servers or a view change takes place.

Proof: According to Obs 1 a correct representative will not assign two different updates to the same sequence number. If P1 is invoked on some update by a correct representative, then the representative assigns some sequence number for that update. If P1 does not terminate at at least $f+1$ correct servers,

then these servers will see no proggess being made for that sequence number and they will invke a view change. Since $f+1$ servers are enough to create a view change, then a view change will take place. ■

Claim: If P1 returns (d, s) at any correct server, then P1 will never return (even in a different view) at any correct server (d', s) with $d' \neq d$.

Proof: Following Lema 6.4 and Lema 6.7, the Claim is proved. ■

Lemma 6.4: If P1 returns (d, s) in some view v at any correct server, then $f+1$ correct servers have a Prepare-Certificate for (d, s) in the view P1 returned.

Proof: According to the rules RL3 and RL5, a correct server adopts a Prepare Certificate for a sequence (d, s) either when it does not have one already, or when it has one from an older view. In any case, the old certificae (if any) is replaced with the new one. Therefore, a correct server cannot have two Prepare-Certificates for the same sequence number.

If P1 returns, then P2 (as part of P1) was invoked at 2^*f+1 servers. Out of these, at least $f+1$ are correct. This implies that at least $f+1$ correct servers should have constructed a Prepare-Certificate out of the L-Pre-Prepare and the set of 2^*f L-Prepare messages that served as a pre-condition to invoking P2. ■

Lemma 6.5: If $f+1$ correct servers have a Prepare-Certificate for (d, s) in some view v , and a view change takes place, then any correct representative of the next view installed after v will not issue an L-Pre-Prepare message contianing a different data d' and s .

Proof: Any set of 2^*f+1 PrepareOK messages in P4 in the same view will contain at least one server of the set of $f+1$ having the Prepare-Certificate for (d, s). Therefore, any correct representative elected will apply to its Context.Local_History a Prepare-Certificate for (d, s) in the new view $v' \geq v$, and consequently will not send an L-Pre-Prepare message containing d' and s , with $d' \neq d$. ■

Lemma 6.6: If $f+1$ correct servers have a Prepare-Certificate for (d, s) in some view v , they will all have a Prepare-Certificate for (d, s) in any view higher than v .

Proof: According to rule RL5 a Prepare-Certificate(d, s, v) can only be replaced by another Prepare-Certificate(d', s, v') constructed in a higher view $v' > v$. In order for such a new certificate to be constructed, at least 2^*f servers, in addition to the representative, should send a L-Prepare(d', s, v') message. However, since $f+1$ correct servers already have a Prepare-Certificate for (d, s), they will not send an L-Prepare message as their Conflict() function will return TRUE (Lines B8-B9 in Conflict()). Therefore, there cannot be a set of 2^*f servers, in addition to the representative, sending a L-Prepare(d', s, v'), ■

Lemma 6.7: If $f+1$ correct servers have a Prepare-Certificate for (d, s) in some view v , then P1 cannot return (d', s), with $d' \neq d$ on any view $v' \geq v$

Proof: According to Lema 2 and Lemma 3, only a corrupt representative could send a L-Pre-Prepare message contianing d' and s with $d' \neq d$. Then, there will be $f+1$ correct

servers, none of them being the representative, that will have a Prepare-Certificate for (d, s) . Following Obs 3, these servers will not send a L-Prepare message for (d', s) , and therefore no other server will receive $2f$ L-Prepare message for (d', s) . As a consequence, there will not be enough servers invoking P2 to complete with result (d', s) . ■

Claim: If a correct representative invokes P1 with some data d and a sequence number s , in a view v , and the preconditions 1-3 are valid, then P1 returns the data d with the sequence number s and a valid threshold signature at $2f+1$ correct servers.

Precondition 1: There are $2f+1$ correct connected servers that have preinstalled the same view v and that do not suspect the representative.

Precondition 2: There is enough time for the necessary communication consisting in three network crossings in protocol P1 to complete before a correct server suspects the representative.

Precondition 3: The data structures in the correct servers have been synchronized so that there are no conflicts. No correct server has a Prepare-certificate, for sequence number s , that the representative does not have, or that was constructed in a more recent view than the one the representative has.

Proof: All correct connected servers will receive Pre-Prepare(d, s, v) from the representative. Following from lines B2-B3 of ASSIGN-SEQUENCE, all correct connected servers will send a matching Prepare(d, s, v) because no conflict occurs. Therefore, all correct servers will receive $2f$ Prepare(d, s, v) messages and 1 Pre-Prepare(d, s, v) message which forms a Prepare_Certificate(d, s, v). At this point, all correct servers will invoke P2. According to Property 1 of P2, the protocol eventually returns to every correct nodes with the combined signature of the data. ■

C. Proof of SITE-ATTEMPT-WAN-VIEW-CHANGE

We say that a server *globally_attempts* a global view v when the server sets its *My_global_view* variable to v . Protocol is presented in Figure 10

Claim: If $2f+1$ correct servers invoke P3, and all correct servers are connected, then P3 eventually returns to every correct server a single view number and a combined signature of that view number.

Claim: P3 will not return to any server a correct combined signature unless at least $f+1$ correct servers that invoked P3 returned the same view number.

PROPERTY 6.1: If $2f+1$ correct servers within a site are connected, they will either all make progress, or they will eventually all Globally_attempt the same global view v , and they will all generate a Global_VC message for v .

PROPERTY 6.2: If $2f+1$ correct servers within a site are connected, and these servers have all *globally_attempted* the same view, v , then if global progress is not made, these correct, connected servers will all *globally_attempt* view $v+1$, and will generate a Global_VC message for view $v+1$.

PROPERTY 6.3: If $2f+1$ correct, connected servers within a site invoke P3 in the same *globally_attempted* view v , then all correct servers will generate a threshold-signed Global_VC message for view v .

Proof: All correct servers send L_VC_Attempt messages for view v . Since all correct servers are connected, they all receive at least $2f+1$ L_VC_Attempt messages for view v . Then, in lines E2 and E3, they all generate threshold signature shares for view v , and send these shares to all local servers. All local servers receive the shares, combine them, and return the same Global_VC message at line F4.

Since at least $2f+1$ correct servers invoked P3 from global view v , no other server can have *global_attempt_proof* for a view higher than v . Thus, no correct server will move beyond v in this invocation. ■

PROPERTY 6.4: Property 2: If $2f+1$ correct, connected servers within a site, having the same value for *Global_view*, invoke P3, then the site will eventually generate a Global_VC message.

Proof: Since $2f+1$ correct servers invoke P3 from the same global view, each such correct server will receive $2f+1$ distinct, L_VC_Attempt signature shares for its own view. The servers then combine in line D3, and return the Global_VC message. ■

D. Proof of CONSTRUCT-COLLECTIVE-STATE

Claim: If a correct representative invokes P4 with some sequence number and all correct servers are connected, then P4 will return a set of data updates, each with a corresponding sequence number higher than the one invoked, and a proof of the set, or a view change takes place.

Lemma 6.8: If a correct representative invokes P4 with sequence number seq , and all correct servers are connected, then the representative will receive at least $2f+1$ Server_Set messages from the same view in response to its initial message, or a view change takes place.

Proof: By assumption, there are at least $2f+1$ connected, correct servers when P4 is invoked. These servers will all receive the representative's initial invocation message at line B1.

If a correct server is in a different view than the representative, it will not respond to the invocation message. If there exists at least one such server, then since all faulty servers may refuse to respond, either the representative receives $2f+1$ Server_Set responses from some combination of correct and faulty servers, or a view change occurs because no progress is made.

Assume, then, that all correct servers are in the same view as the representative. Then upon receiving the representative's invocation message, a correct server will either enter case (1) or case (2), at lines B4 and B7, respectively.

If a server enters case (2), it sends a Server_History message to the representative immediately.

If the server enters case (1), then the server has a local aru, l_{aru} , lower than seq , and requests those updates between l_{aru} and seq (line B5). Since there are no communication failures,

the server is able to eventually recover these updates from the representative. When this recovery completes, the server enters case (2), constructs a Server_Set message, and sends it to the representative.

Since we assume in this case that there are at least $2f+1$ correct servers in the same view as the representative, either at least $2f+1$ Server_History messages arrive at the representative, or a view change occurs. ■

A server constructs a Server_Set message by calling Construct_ServerSet(seq). The Server_Set message contains a (possibly empty) set of items. Any item that does appear is of one of the following two types:

- 1) Proposal
- 2) (Proposal, Accept_List)

We say that an item of type (1) is valid if:

- The Proposal carries a valid threshold-signature from site_id.
- $\text{prop_global_seq} > \text{seq}$

We say that an item of type (2) is valid if:

- The Proposal carries a valid threshold-signature from site_id.
- $\text{prop_global_seq} > \text{seq}$
- For each Accept A in Accept_List:

- 1) A carries a valid threshold-signature from site A.site_id
- 2) $\text{A.accept_global_seq} == \text{prop_global_seq}$
- 3) $\text{A.digest} = \text{digest}(\text{Proposal.Update})$
- 4) $|\text{Accept_List}| == (\text{Majority} - 1)$
- 5) $\text{Proposal.site_id} != \text{A.site_id}$ for any A in Accept_List
- 6) $\text{A1.site_id} != \text{A2.site_id}$ for any A1, A2 in Accept_List
- 7) $\text{A1.view_num} == \text{A2.view_num}$ for any A1, A2, in Accept_List
- 8) $\text{Proposal.view_num} == \text{A.view_num}$ for any A in list

We say that a Server_Set message is valid if any item that appears is valid. Lemma 6.9 states that any Server_Set message constructed by a correct server is valid.

Lemma 6.9: Let M be the Server_Set message constructed by correct server s in response to a correct representative's invocation message with sequence number seq. Then M is a valid Server_Set message.

Proof: A correct server constructs a Server_Set message by calling Construct_ServerSet(seq). For any sequence number $s > \text{seq}$, if $\text{Global_History}[s]$ is empty, then the server will not add any item to the Server_Set message for this sequence number, which trivially meets the validity properties defined above.

For any sequence number $s > \text{seq}$ where $\text{Global_History}[s]$ is not empty, if the server has globally ordered s, i.e. if $\text{Global_Ordered_Ready}(s) == \text{true}$, then it includes the Proposal and its corresponding Accept_List. This results in an item of type (2). By the rules for global ordering, this item is valid.

If the server has not globally ordered s, then $\text{Global_History}[s]$ may consist of (a) Only a Proposal or (b) A Proposal and $1 \leq j < (\text{Majority}-1)$ Accepts. The server adds the Proposal to the Server_Set message in both cases as an item of type (1). By the rules for maintaining the Global_History, such an item is valid.

Since these are the only two type of items that may appear in the Server_Set message, then the Server_Set message M of server s is valid. ■

Lemma 6.10: If a correct representative invokes P4 with sequence number seq, and all correct servers are connected, then the representative will receive at least $2f+1$ valid Server_Set messages from its own view in response to its initial message, or a view change occurs.

Proof: From Lemma 6.8, the representative receives at least $2f+1$ Server_Set messages in response to its initial invocation message, or a view change occurs. From Lemma 6.9, any Server_Set message sent by a correct server is valid.

If any correct server does not respond to the representative's initial invocation message (because it is in a different view), then either the representative receives $2f+1$ valid Server_Set messages (from some combination of correct and faulty servers) or a view change occurs.

If all correct servers send Server_Set messages, then either the representative receives a set of $2f+1$ valid Server_Set messages (either from all correct servers or some combination of correct and faulty servers who send valid messages), or a view change occurs before these messages arrive. ■

Lemma 6.11: If a correct representative invokes P4 with sequence number seq, and all correct servers are connected, then at least $f+1$ correct servers will receive and process a CollectedServerHistorySet message from the representative's view, or a view change occurs.

Proof: By Lemma 6.10, the representative receives at least $2f+1$ valid Server_Set messages from its own view, or a view change occurs. Since there are at most f faulty server, at least $f+1$ of these valid messages are from correct servers in the same view as the representative.

The representative sends a CollectedServerHistorySet message to all local servers. Since all correct servers are connected, the representative's message will arrive at all correct servers, including the $f+1$ whose Server_Set messages the representative received. ■

Lemma 6.12: If a correct representative invokes P4 with sequence number seq, and all correct servers are connected, then at least $f+1$ correct servers return identical (ConstraintMessage, AruMessage) pairs as the result of P4, or a view change occurs.

Proof: By Lemma 6.11, at least $f+1$ correct servers receive (and process) a CollectedServerHistorySet from the representative, or a view change occurs. Since the representative is correct, all correct servers receive identical CollectedServerHistorySet messages. Since the representative's signature cannot be forged, a correct server will only process CollectedServerHistorySet message from the representative.

Each of these $f+1$ correct servers produces the union in a deterministic way, based solely on the contents of the CollectedServerHistorySet message. Thus, each produces the same union. At this point, at least $f+1$ correct servers construct the ConstraintMessage by invoking Threshold_Signature on the identical union.

If even one correct server is not in the same view as the representative, then either Threshold_Signature terminates with some combination of at least $f+1$ correct servers and some faulty servers (in which case the $f+1$ correct servers return identical ConstraintMessages), or it fails to terminate, and a view change occurs.

If all correct servers are in the same view as the representative, then they all invoke Threshold_Signature on the same union, and produce identical ConstraintMessages, or a view change occurs.

Similarly, each of these servers produces union_aru in a deterministic fashion, and invokes Threshold_Signature on this result. By the same property, each such AruMessage is identical.

In either case, at least $f+1$ correct servers return identical (ConstraintMessage, AruMessage) pairs, or a view change occurs. The threshold signature attached to each message serves as proof that the site assented to the message. ■

Lemma 6.13: If a correct representative invokes P4 with sequence number seq, and all correct servers are connected, then any item contained in the ConstraintMessage returned by at least $f+1$ correct servers will be for a sequence number greater than seq, or a view change occurs.

Proof: By Lemma 6.12 , at least $f+1$ correct servers return the same ConstraintMessage, or a view change occurs. The ConstraintMessage is the result of invoking Threshold_Signature on the computed Union.

The Union function operates on the ServerHistorySet messages contained in the CollectedServerHistorySet message. By the way in which the representative constructs the CollectedServerHistorySet message, each such ServerHistorySet message is valid. By the properties of validity, any item contained in a valid ServerHistorySet message is for a sequence number greater than seq. Since the result of the Union function is a subset of those items appearing in the ServerHistorySet messages, any item remaining must be for a sequence number greater than seq. ■

Claim: Claim 2: Any (d, s) in the ConstraintMessage returned by P4 passes context verification:

- 1) For the Global_History, (d, s) was Proposed by some leader site.
- 2) For Pending proposals, (d, s) has a Prepare Certificate

Proof: If a correct server returns from P4, then it successfully completes the Union() function. This means that each of the $2f+1$ ServerHistorySet messages contained in the CollectedServerHistorySet message is authenticated.

A ServerHistorySet message will only be authenticated if:
(1) The message carries a valid signature from a server or (2)
Each item (if any) contained in the message carries a valid

threshold signature, and is for a sequence number greater than seq.

An item of type (1) is a Proposal, and since it was authenticated, must have been proposed by some leader site, since it carries a valid threshold signature. Such a message was accepted by some server in the site, since the server stored it in its history.

An item of type (2) consists of a Proposal and (Majority-1) corresponding Accepts. Since the ServerHistorySet message is valid, the item passes the validity tests defined above. Then the authentication confirms that the Proposal and its Accepts are valid. If the authentication succeeds, then this item constitutes proof that the message was globally ordered by the server who included it in its ServerHistorySet message.

The Union() returns a subset of the items contained in these valid, authenticated ServerHistorySet messages, and thus the ConstraintMessage produced as the result of P4 will consist only of items that were either globally ordered or accepted by some server in the site. ■

Claim: Claim 3: Let s be the sequence number with which P4 was invoked. Then for any (d, s') , where $s' > s$,

- 1) If (d, s) was globally ordered by at least $f+1$ correct servers in the site, then it will appear in the ConstraintMessage as globally ordered.
- 2) If (d, s) was in Global_History[s] as a Proposal at at least $f+1$ correct servers, then it will appear in the ConstraintMessage as a Proposal, unless it could not have been globally ordered.

Lemma 6.14: Let s be the sequence number with which P4 was invoked. Then any correct server that responds to the initial invocation message and that has either globally ordered (d, s') with $s' > s$, or accepted (d, s') with $s' > s$, will include (d, s') in its ServerHistorySet message.

Proof: If a correct server has accepted (d, s') with $s' > s$ but has not globally ordered it, then by the protocol, the server includes a type (1) item for (d, s') in its ServerHistorySet message. If the server has globally ordered (d, s') with $s' > s$, then by the protocol, the server includes a type (2) item in its ServerHistorySet message corresponding to (d, s') . In both cases, the item appears in the ServerHistorySet message, completing the proof. ■

Lemma 6.15: Let s be the sequence number with which P4 was invoked. Then any (d, s') , where $s' > s$, that was either globally ordered by at least $f+1$ correct servers in the site, or accepted by at least $f+1$ correct servers in the site, will appear in at least one of the ServerHistorySet messages contained in authenticated, valid CollectedServerHistory message.

Proof: The CollectedServerHistorySet message consists of $2f+1$ ServerHistorySet messages from distinct servers. Since there are at most f faulty servers within the site, at least $f+1$ of these messages are from correct servers.

By assumption, (d, s') was either globally ordered by at least $f+1$ correct servers, or accepted by at least $f+1$ correct servers. Then in any set of $2f+1$ ServerHistorySet messages, at least one ServerHistorySet message is from one of these

$f+1$ correct servers. By Lemma 1, such a ServerHistorySet message includes (d, s') . ■

Lemma 6.16: Let (d, s, v) be an item that appears in one of the $2f+1$ valid ServerHistorySet messages used in Union(). Then (d, s, v) either appears in the set returned by Union, or the set returned by Union contains (d', s, v') such that $v' > v$.

Proof: (d, s, v) appears as a type (2) item in one of the ServerHistorySet messages if it was globally ordered by that server. By the Union function, type (2) items are only removed from the resulting union if an item corresponding to this globally ordered update already exists in the Union. Thus, such an item will appear in the o_union set.

(d, s, v) appears as a type (1) item in one of the ServerHistorySet messages if it was accepted by that server. The a_union set is constructed by combining all accepted items from the $2f+1$ messages, and removing identical items. If there exists a (d', s, v') such that $v' > v$, then (d, s, v) is replaced by (d', s, v') . If $d' == d$, then (d', s, v) represents the same Proposal from a later view. If $d' \neq d$, then (d', s, v) represents a different Proposal from a later view, which can only occur if the (d, s, v) was not globally ordered. ■

E. Proof of the LAN-VIEW-CHANGE Protocol

In this section we prove the algorithm responsible of changing the view within a site and electing a new representative. We have specified two functionally equivalent algorithms. The first protocol was specified in Figure 9. In this section, we prove properties of the second protocol which is specified in Figure 16.

We use the following definitions:

DEFINITION 6.1: Preinstall; We say that a server preinstalls view v if it collects $2f+1$ New_Rep messages for a view equal to or greater than v . It can collect these messages via two different mechanisms: 1. It can receive the messages directly from the servers that generated them. 2. It can receive a Preinstall_proof message that contains a set of $2f+1$ New_Rep messages where the minimum New_Rep message is for view v .

DEFINITION 6.2: Progress; From an individual server's view, progress means that any update which a server has sent to the representative for ordering has been globally ordered within some expected amount of time. If the update is not eventually globally ordered, all correct connected servers will suspect the representative.

Each server maintains four variables related to the LAN-VIEW-CHANGE protocol. They are:

- $\text{IAm_Representative}$: 1 if this server is the representative for this site
- Representative: the id of the representative for this site
- Local_view : the local view number of this server
- $\text{Installed_local_view}$: 0 when the view change protocol is invoked; set to 1 when the protocol ends

When a correct server receives an update, it puts this update in its queue. It removes updates from the queue when they have been ordered. The protocol is triggered by the following events:

- Any new update is not globally ordered within a timeout period, Δ . This update may or may not be an update that is in the servers Update_Queue .
- The oldest (first) update in the servers Update_Queue is not ordered within a timeout period greater than Δ .

We assume that the following precondition is true:

PRECONDITION 6.1: The initial Local_view , in which all correct servers start, is preinstalled, a priori. Therefore all servers have an a priori preinstall_proof for this view.

We can now prove the following lemma.

Lemma 6.17: Any correct server with its Local_view equal to v , has either preinstalled view v or preinstalled view $v-1$.

Proof:

A correct server can increase its view only in the following cases:

- The server responds to a trigger event (this includes timeouts) and increments its Local_view by one.
- the server receives a proof that a view preinstalled (the proof consists of a set of $2f+1$ L_New_Rep messages). If it receives this proof, it preinstalls view V where V is the lowest L_New_Rep .

Following from Precondition 6.1, servers that have not yet incremented their view have preinstalled the initial view (and have a preinstall_proof), a priori. From the algorithm, a correct server cannot respond to a trigger event and increment its view unless it has preinstalled the view from which it is moving. Therefore, when a server increases its Local_view , it must either be responding to a preinstall_proof , in which case it preinstalls the view to which it is moving, v , or it must be responding to a trigger event, in which case, if it moves to view v , it has preinstalled view $v-1$. ■

We can now prove the properties of the LAN-VIEW-CHANGE protocol.

PROPERTY 6.5: If $2f+1$ correct servers are connected, they will either make progress or they will eventually all preinstall the same view.

Proof: By Lemma 6.17, if a correct server responds to a trigger event and increments its Local_view to v , it must have a preinstall_proof for view $V-1$. If progress is not made, there will be trigger events at all correct servers. Suppose the maximum preinstalled view is M . Let Max_Server denote a server with $\text{Local_view} == M$. When a trigger event occurs at Max_Server , it will send a New_Rep message for view $M+1$ to all correct servers. Consider any correct server, S . If the maximum view that S has preinstalled is less than M , then S will request a preinstall_proof from Max_Server . When S receives the preinstall_proof for M , S preinstalls view M and sets its Local_view to M . At this point, if progress is not made, there will be a trigger even at S and S will increment its view to $M+1$. Therefore, if progress is not made, all correct servers will increment their view to $M+1$. At this point, there are at

least $2f+1$ servers that will send a `New_Rep` message for view $M+1$ and view $M+1$ will be preinstalled. ■

PROPERTY 6.6: If $2f+1$ correct servers are connected and these servers have all preinstalled the same view, v , then if progress is not made, these correct connected servers will all preinstall view $v+1$.

Proof: If at least $2f+1$ correct connected servers have preinstalled view v , then if progress is not made, all correct connected servers will either undergo a trigger event or preinstall $v+1$ because they receive a preinstall proof. When a server responds to a trigger event it will increase its view by 1 to view $v+1$. As soon as the first server preinstalls view $v+1$, it can prove to the other correct connected servers that view $v+1$ was preinstalled, and they will all preinstall $v+1$. If all correct servers have the same `Local_View`, V , the f faulty servers cannot prove that any view greater than v has preinstalled. Therefore, a correct server that times out must preinstall view $v+1$ before preinstalling a higher view. ■

PROPERTY 6.7: If $2f+1$ correct servers are connected, then if $f+1$ correct servers learn about an update and do not receive proof that the site initiated that update after some amount of time, there will be a view change.

Proof: When a correct server receives an update, it sends the update to all local servers and each of these servers sends it to the representative. Therefore, when any correct server learns of an update, all correct connected servers will learn of this update. When a server learns of an update, it sets a timeout. If this timeout expires before the server receives proof that the site initiated the update, it suspects the representative.

If $f+1$ servers suspect the representative, then they will all timeout and increment their `Local_View`. These servers will not participate in `ASSIGN-SEQUENCE` protocol until they preinstall a higher view. Since $2f+1$ servers must participate in `ASSIGN-SEQUENCE` to make progress, the remaining f correct servers will also timeout and attempt to preinstall another view. ■

F. Local L1 (Local Progress)

If there exists a set, within a site, consisting of at least $2f+1$ correct servers, and a time after which the correct members of this set are connected, then if a server in the set initiates an update, the site will eventually initiate the update.

PRECONDITION 6.2: In the case where Local L1 refers to the pending context (which runs only on the leader site and produces Paxos Proposals), the global constraint messages required for `ASSIGN-SEQUENCE` to make progress generating global Proposals have already been received.

Definitions:

DEFINITION 6.3: `site_update_progress`; We define `site_update_progress` to mean that when a correct server initiates an update, the site will initiate this update.

DEFINITION 6.4: `site_update_progress_proof`;

- If all connected servers have proof that `site_update_progress` is made `site_update_progress_proof` is True

- If no connected servers have proof that `site_update_progress` is made `site_update_progress_proof` is False
- Otherwise `site_update_progress` is undefined.

DEFINITION 6.5: `initiate`:

- We say a server initiates an update when the server receives an update that should be locally ordered and attempts to push the update into the system.
- We say a site initiates an update when the update is sent out of the site to the representative of the leader site.

DEFINITION 6.6: `blocking_conflict`; A `blocking_conflict` is a situation where a conflict occurs that cannot necessarily be resolved without a view change. Thus, a `blocking_conflict` can potentially require a view change. We say that a conflict occurs when the conflict function is called on a `Pre-Prepare` and returns true.

Proof of Local L1:

Lemma 6.18: If $2f+1$ correct servers are connected and `site_update_progress_proof` is False, $2f+1$ correct servers will eventually all preinstall the same view.

Proof: From Definition 6.3, no correct server receives proof that `site_update_progress` occurred. Lemma 6.18 follows from Proof of Local Representative Election Protocol Property 1 and Property 3. ■

Lemma 6.19: If `site_update_progress_proof` is True, then Local L1 is true.

Proof: `Site_update_progress_proof` can be True only if `site_update_progress` is made. If there exists proof that site update progress was made, then the site must eventually initiate an update if a correct server initiated this update. `Site_update_progress` implies Local L1, by Definition 6.4. ■

Lemma 6.20: If there are $2f+1$ correct connected servers and `site_update_progress_proof` is False, every correct server will eventually often become the representative. Each correct server will become representative once every N view changes where N is the number of servers in the site.

Proof: Following from Lemma 6.19, if `site_update_progress_proof` is False, all correct servers will preinstall the same view. By Local Leader Election Property 2 and Property 3, if at least $2f+1$ correct connected servers preinstall the same view, V , and `site_update_progress_proof` is False, then these servers will all preinstall view $V+1$. When all servers preinstall $V+1$, then if `site_update_progress` is not made, this process repeats and they will all preinstall $V+2$. Therefore, views are preinstalled consecutively. Because we use V representative every N view changes where there are N local servers. Thus, all correct servers will become the representative for some amount of time, Delta, every N view changes. ■

Lemma 6.21: If there are $2f+1$ correct connected servers, $P4$ will eventually complete.

Proof: By `New_Property_of_P4` (see below), $P4$ will return if a correct representative invokes $P4$ and the preconditions for this property are met. Precondition 1 is met because we have $2f+1$ correct connected servers and Lemma 3 says that

```

Initial State:
Local_view = 0
my_preinstall_proof = a priori proof that view 0 was preinstalled
Set Timer Local_T = L_Expiration_Time

LAN-VIEW-CHANGE()
A1. if ( my_preinstall_proof.preinstalled_view ≠ Local_view )
A2. return
A3. Local_view++
A4. Stop Timer Local_T
A5. L_Expiration_Time *= 2
A6. l_new_rep <- Construct_New_Rep(My_server_id, Local_view)
A7. Send to all local servers: L_New_Rep

B1. Upon Local_T expiration:
B2. LAN-VIEW-CHANGE()

C1. Upon a trigger which causes me to suspect my representative:
C2. LAN-VIEW-CHANGE()

D1. Upon receiving an L_New_Rep message from server S:
D2. if L_New_Rep.view > Local_view + 1,
D3. request <- Construct_Preinstall_Proof_Request_Message()
D4. Send to server S: request

E1. Upon receiving a request for a preinstall_proof from server S:
E2. Send my_preinstall_proof to S

F1. Upon receiving a preinstall_proof my message, p, for view V:
F2. if Local_view < my_preinstall_proof.preinstalled_view
F3. my_preinstall_proof <- p
F4. Local_view <- p.preinstalled_view
F5. Set Time Local_T = L_Expiration_Time

G1. Upon receiving a set, L_New_Rep_Messages, of 2f+1 distinct L_New_Rep
G2. for a view greater than or equal to mine:
G3. my_preinstall_proof <- Construct_Preinstall_Proof(L_New_Rep_Messages)
G4. if not new representative
G5. Set Timer Local_T = L_Expiration_Time
G6. if new representative
G7. Send to all local servers: my_preinstall_proof
G8. union <- Invoke CONSTRUCT-COLLECTIVE-STATE(L_Aru, Local_Update_History)
G9. //Process union, decide what to replay
G10. Invoke ASSIGN-SEQUENCE for each unordered update

```

Fig. 16. Version 2 of LAN-VIEW-CHANGE Protocol.

if site_update_progress_proof is False, the $2f+1$ correct connected servers will preinstall the same view. If view changes occur during P4, by Lemma 4, eventually often a correct server will invoke P4 again. If site_update_progress_proof is False, then the timeout expiration increases so that eventually it will become large enough to meet Precondition 2. ■

Lemma 6.22: When a correct server invokes ASSIGN-SEQUENCE on sequence number s and a blocking_conflict occurs, there is at least one correct server that can invoke ASSIGN-SEQUENCE with sequence number s such that another blocking_conflict cannot occur unless there exists a distinct Prepare_Certificate for sequence number s that has not previously caused a blocking conflict.

Proof: Let $S1$ and $S2$ denote two correct servers which have preinstalled the same view, have completed P4, and do not suspect their representative. Suppose that $S1$ is the representative. $S1$ sends Pre-Prepare(d, s, v) and $S2$ receives this Pre-Prepare. Following from lines B1 to B12 of the conflict function, when $S2$ calls the conflict(Pre-Prepare(d, s, v)) it can return True in two cases:

- Conflict-1: $S2$ has Prepare_Certificate(d', s, v') where $d' \neq d$

- Conflict-2: $S2$ has Local_Ordered_Update(d', s, v') where $d' \neq d$ OR $v' > v$

If $S2$ calls the conflict function and it returns True, $S2$ will not send a Prepare message immediately. Instead, $S2$ will table the Pre-Prepare message and request from $S1$ the data upon which the Pre-Prepare message was based. This request message includes the source of the conflict which is either a Prepare_Certificate or a Local_Ordered_Update. A Pre-Prepare(d, s, v) is based on either no prepare certificate or on a Prepare_Certificate(d, s, v').

In case Conflict-1 the following six scenarios exist. The scenarios that can result in a view change (block) are marked with BLOCKING. As stated above, $S1$ sent Pre-Prepare(d, s, v) and $S2$ had a Prepare-Certificate(d', s, v').

- 1) If $S1$ has a Prepare_Certificate(d, s, v'') and $v' < v''$, then $S2$ (through gossip) will apply this Prepare_Certificate to its data structures because it has a higher view than the Prepare_Certificate that $S2$ already had. Then $S2$ can send Prepare(d, s, v). In this scenario, a conflict resolution takes place.

- 2) BLOCKING If S1 has a `Prepare_Certificate(d,s,v")` and $v' > v"$, then S1 will adopt the `Prepare_Certificate` from S2. In this case, there is a possibility of a block because S2 will not send `Prepare(d,s,v)`.
- 3) BLOCKING If S1 has no `Prepare_Certificate` for sequence number s, then S1 will adopt the `Prepare_Certificate` from S2. In this case, there is a possibility of a block because S2 will not send `Prepare(d,s,v)`. Note that during the replay phase of a view change the data in this scenario is often a NOP.
- 4) If S1 has a `Local_Ordered_Update(d",s,v")`, then S1 will send this to all local servers. Note that in this case, sequence number s has already been bound to some update. Therefore, `ASSIGN-SEQUENCE` already completed.
- 5) BLOCKING If S1 has a `Prepare_Certificate(d",s,v")` where $d \neq d"$ and $v' > v"$, then S1 will will adopt the `Prepare_Certificate` sent by S2. There can be a block in this case because S2 will not send `Prepare(d,s,v)`.
- 6) BLOCKING If S1 has a `Prepare_Certificate(d",s,v")` where $d \neq d"$ and $v' < v"$, then S2 will adopt this `Prepare_Certificate`. There can be a block in this case because S2 will not send `Prepare(d,s,v)`.

Note that the scenarios 4, 5, and 6 can occur only when S1 applies a change to its data structure after it sent `Pre-Prepare(d,s,v)`.

In case Conflict-2 the following scenario exists: S2 will send the `Ordered_Update` to S1. If S1 has not already received an `Ordered_Update` with a greater view, it will apply the `Ordered_Update` that S2 sent. At this point, since sequence number s has already been bound to data, we consider `ASSIGN-SEQUENCE` to have completed for sequence number s.

If there is a blocking conflict `ASSIGN-SEQUENCE` may not complete and then there will be a view change. The only conflicts that may cause a block are scenarios 2,3,5, and 6. All other scenarios have conflicts which can be resolved without a view change by updating the servers with conflicts. Following from Lemma 6.20, if a block does occur and `site_update_progress` is not made, eventually the server that has the conflicting data, S2, will become the representative.

Now consider what happens when S2 becomes the representative during each of the possible blocking scenarios. It is possible that S2 has received and applied a new `Prepare_Certificate` or `Local_Ordered_Update` to its datastructures since the conflict occurred. If S2 has a `Local_Ordered_Update` for sequence number s, then `ASSIGN-SEQUENCE` has completed. If S2 has a new `Prepare_Certificate`, then this `Prepare_Certificate` must have a higher view than the `Prepare_Certificate` that caused the blocking scenario or it would not be applied to the data structures (follows from Rule RL5).

In each of the four scenarios that cause blocks and therefore view changes, S2 will eventually become the representative if `site_update_progress` is not made. The following describe what happens when S2 is the representative:

- 1) Scenario 2: For sequence number s, S2 either has a `Prepare_Certificate` with view v' or higher or Lo-

`cal_Ordered_Update`. Note that v' was greater than v . If S2 has a `Prepare_Certificate`, the view number of this `Prepare_Certificate` must be greater than the view number of the `Prepare_Certificate` that resulted in the blocking condition when S1 sent `Pre-Prepare(d,s,v)`.

- 2) Scenario 3: S2 is the representative. S2 has a `Prepare_Certificate` or an `Ordered_Update` for sequence number s. Originally, S1 had no `Prepare_Certificate` for sequence number s.
- 3) Scenario 5: This is the same as for scenario 1.2.
- 4) Scenario 6: For sequence number s, S2 either has a `Prepare_Certificate` with a view higher than v' or `Local_Ordered_Update`. Note that v' was greater than v . If S2 has a `Prepare_Certificate`, the view number of this `Prepare_Certificate` must be greater than the view number of the `Prepare_Certificate` that resulted in the blocking condition when S1 sent `Pre-Prepare(d,s,v)`.

In case 1,3, and 4 above, the sequence number of S2's `Prepare_Certificate` is greater than the sequence number of the `Prepare_Certificate` that S1 had when S1 originally sent `Pre-Prepare(d,s,v)`. In case 2, S1 had no `Prepare_Certificate` and S2 had a `Prepare_Certificate`. During subsequent blocking conditions, all scenarios above can happen again except for scenario 1.3. In each of the three remaining blocking scenarios, a server that has a conflict must have a `Prepare_Certificate` with a view greater than the `Prepare_Certificate` that the representative has. Therefore, each time a block occurs there is at least one correct representative which will invoke `ASSIGN-SEQUENCE` on sequence number s with data such that there cannot be another blocking scenario unless there is a distinct `Prepare_Certificate` that has never caused a blocking scenario. ■

Lemma 6.23: If a correct server invokes `ASSIGN-SEQUENCE` on sequence number s, then `ASSIGN-SEQUENCE` will eventually return.

Proof: There are a finite number of `Prepare_Certificates` for the same sequence number and different views in the system (this includes those `Prepare_Certificates` known only to faulty servers). Following from 6.22, eventually a correct server will be representative and will base its invocation of `ASSIGN-SEQUENCE` on the `Prepare_Certificate` for sequence number s with the highest view. Through the described gossiping mechanism, any conflicts can be resolved if the representative sends its `Prepare_Certificate` to all servers. Following from P1 Property 3, this invocation will complete because there will eventually be no conflicts.

A finite number of `Prepare Certificates` for the same sequence number, different data, and different views can exist: A bad representative cannot generate a prepare certificate without the cooperation of at least $f+1$ good servers. Suppose in view 1 $PC(1, d1)$ is generated. In each subsequent view where there is a bad representative, another PC can be generated with the same sequence number and different data. This can occur f times before there will be a good server. When there is a good server, the bad servers can give one of the correct no-reps a PC. This can cause a block. However, one correct server now

has a PC. It will give this PC to all servers when it becomes the representative. A server will not contribute a Prepare for another PC with different data unless it receives a PC having a higher view than the view that it has. ■

Lemma 6.24: If there are $2f+1$ correct connected servers, eventually a representative will complete the replay phase of the local view change algorithm.

Proof: By Lemma 6.21, P4 will eventually complete. By ASSIGN-SEQUENCE Property 3, for any sequence number that needs to be replayed, a correct representative will eventually complete ASSIGN-SEQUENCE. There is a finite range of sequence numbers that must be replayed. Therefore, the replay phase will complete.

The following uses a window to argue that there is a bound on the number of sequence numbers that need to be replayed.

If a correct server sends a Pre-Prepare in response to a Prepare for sequence number s , it must have proof that all sequence numbers up to $s-W$ have been locally ordered. $2f+1$ servers must send Prepare or Pre-Prepare messages to create a Prepare_Certificate or a Local_Ordered_Update. Therefore, if there exists a Prepare_Certificate or Local_Ordered_Update for sequence number s , $f+1$ correct servers must have proof that all updates up to $s-W$ were locally ordered. In this case, the replay window will begin no earlier than $s-W$ and can end no later than s . Therefore, the number of sequence numbers that must be replayed is less than W . Because there are a finite number of sequence numbers that need to be replayed, eventually the replay phase will finish.

The window mechanism used in the above proof prevents malicious servers from collecting some large number of Prepare_Certificates which good servers know nothing about (As described below). ■

We can now prove Local L1.

Proof: By Lemma 6.19, Local L1 holds if site_update_progress_proof is True. If site_update_progress is not made, by Lemma 6.24, a correct representative will finish the replay process. When the replay process is finished, a correct representative can order new updates. By Lemma 6.20, if site_update_progress_proof is false, eventually often there will be a correct representative. If there is a correct representative that orders an update, this representative can send ordered updates out of the site. Therefore, when a correct server initiates an update, the site will eventually initiate the update. ■

G. Global L1 (Global Progress) Proof

If there exists a set consisting of a majority of sites, each meeting Local_L1, and a time after which all sites in the set are connected, then if a site in the set initiates an update, some site in the set eventually executes the update.

Proof of Global L1: By Claim G.1, either global progress is made or all correct servers in all sites in the majority set eventually reconcile their Global Histories to a common prefix. If no global progress is made by the time this reconciliation completes, then by Claim G.2, all sites in the majority set

eventually preinstall the same global view. By Claim G.3, if all correct servers in all sites in the majority set are reconciled and have preinstalled the same global view, then either global progress is made, or all correct servers in the leader site will be properly constrained by completing the global view change protocol. By Claim G.4, the preconditions for completing Assign-A-Sequence will eventually be met at the leader site such that a valid Proposal can be generated. By Claim G.5, such a Proposal will be able to elicit enough Accept messages for the update to be globally ordered. Definition G.1: The Site-Max-ARU of a site S is the sequence number below which all updates have been globally ordered by one or more correct servers in S.

Claim G.1: If there exists a set consisting of a majority of sites, each meeting Local_L1, and a time after which all sites in the set are connected, then either global progress is made, or eventually all correct servers in all sites in the set reconcile their Global Histories up to highest Site-Max-ARU of any site in the set.

Lemma G.1.1: Let S be a site meeting Local_L1. If no new Proposals are received by any server in S, then all correct servers in S eventually reconcile their Global History to the Site-Max-ARU.

Proof: By Local_L1, there exists a set of $2f+1$ correct, connected servers in S. Since no new Proposals are received in S, the Site-Max-ARU remains fixed. The correct servers continuously run the local reconciliation protocol to exchange globally ordered updates. Since all correct servers are connected, they eventually all reconcile their Global History to the Site-Max-ARU.

Lemma G.1.2: If there exists a set consisting of a majority of sites, each meeting Local_L1, and a time after which all sites in the set are connected, then if no new Proposals are received by any server in any site in the set, all correct servers in all sites in the set eventually reconcile their Global Histories to the highest Site-Max-ARU of any site in the set.

Proof: By Lemma G.1.1, the correct servers in each site in the majority eventually reconcile their Global Histories to their respective Site-Max-ARU values. Each site participates in the global reconciliation protocol to exchange globally ordered updates. Since all sites in the set are connected, the correct servers in each site eventually reconcile their Global Histories to the highest Site-Max-ARU.

Lemma G.1.3: If there exists a set consisting of a majority of sites, each meeting Local_L1, and a time after which all sites in the set are connected, then either global progress is made, or eventually no new Proposals are introduced.

Proof: If no global progress is made by the leader site, there are two cases to consider. In the first case, no global progress is made by any site (i.e. no new updates are globally ordered). In this case, either updates continue to be locally ordered by the leader site until the global window fills up, or no new updates are introduced. In either case, eventually no new Proposals are introduced.

If global progress is made by one or more non-leader sites but not by the leader site, then either the leader site's global

window will fill up, or no new updates are introduced. In both cases, eventually no new Proposals are introduced.

Proof of Claim 1: Immediate from Lemmas G.1.1, G.1.2, and G.1.3.

Claim G.2: If there exists a set consisting of a majority of sites, each meeting Local_L1, and a time after which all sites in the set are connected, then either global progress is made, or all sites eventually preinstall the same global view.

Definition G.2: A site invokes the Site-Attempt-WAN-View-Change protocol when at least $2f+1$ correct servers have invoked the protocol.

Lemma G.2.1: If there exists a set of at least $2f+1$ correct, connected servers within a site meeting Local_L1, then either global progress is made, or the site will eventually invoke Site-Attempt-WAN-View-Change.

Proof: A correct server that has globally attempted view v invokes Site-Attempt-WAN-View-Change when its Leading-Site Timer (T3) expires. When this occurs, the server globally attempts view $v + 1$ and will not actively contribute to generating site messages in view v . Either the remaining servers see global progress, or they, too, will timeout and invoke Site-Attempt-WAN-View-Change, in which case the site is said to have invoked the protocol.

Lemma G.2.2: If there exists a set consisting of a majority of sites, each meeting Local_L1, and a time after which all sites in the set are connected, then either global progress is made, or a majority of sites eventually invoke Site-Attempt-WAN-View-Change.

Proof: By Lemma G.2.1, either global progress is made, or a site meeting Local_L1 will eventually invoke Site-Attempt-WAN-View-Change. Once one site in the set invokes the protocol, either global progress is made without this site, or all sites in the set eventually invoke the protocol. Since the set contains a majority of sites, the lemma holds.

Proof of Claim G.2: By Lemma G.2.2, either global progress is made, or a majority of sites eventually invoke Site-Attempt-WAN-View-Change. By Property 6.1 of the Site-Attempt-WAN-View-Change protocol, if no global progress is made, the correct servers in a site will eventually all globally attempt the same view v and will generate a Global_VC message for view v . By the property of the WAN-View-Change protocol, either progress is made, or a majority of sites eventually preinstall the same global view, completing the proof.

Claim G.3: If there exists a set consisting of a majority of sites, each meeting Local_L1, such that all sites in the set have pre-installed the same global view and have reconciled their Global Histories, then if a correct representative of the leader site sends a Union (Prepare) message in this global view, it will collect a majority of Union (Prepare_OK) responses in this global view, none of which causes a conflict for any correct server in the leader site.

Lemma G.3.1: If there exist at least $2f+1$ correct, connected servers within a site that have preinstalled the same global view v , all of which have reconciled their Global Histories to the Site-Max-ARU, then if Construct-Collective-State is invoked by a correct representative in global view v , all correct

servers in global view v generate the same threshold-signed Union message for global view v .

Proof: By the property of Construct-Collective-State, if the protocol is invoked by a correct representative in a site meeting Local_L1, and all correct servers have reconciled their Global Histories, then the correct servers within a site will complete the Construct-Collective-State protocol in global view v . By the property of Construct-Collective-State, any two correct servers that complete the protocol generate the same threshold-signed Union message for the same view.

Lemma G.3.2: If there exists a set consisting of a majority of sites, each meeting Local_L1, such that all sites in the set have pre-installed the same global view, and all correct servers in all sites in the set have reconciled their Global Histories to the highest Site-Max-ARU of any site in the set, then if a correct representative of the leader site sends a Union (Prepare) message in this global view, it will collect a majority of Union (Prepare_OK) responses in this global view.

Proof: By assumption, there exists a set consisting of a majority of sites that have all preinstalled the same global view and in which global reconciliation has occurred. Since all sites in the set are connected, any non-leader site that has a correct representative will receive the Prepare message from the leader site. The Leading Site timeout (T3) is set such that at least one correct representative will receive the Prepare message in each site in the set. Upon receiving the Prepare message from the currently preinstalled global view, a correct representative of a non-leader site invokes the Construct-Collective-State protocol.

By the property of Construct-Collective-State, at least $2f+1$ correct servers complete the protocol in the preinstalled global view. By the property of Construct-Collective-State, they all generate identical Prepare_OK responses in this global view. The correct representative then sends the Prepare_OK to the representative of the leader site, which, by the relationship between timeouts T1 and T2, is still the same correct server.

Lemma G.3.3: Any item appearing in a Prepare_OK message will not cause any "real" conflict with a correct server at the leader site.

Proof: By Property of Construct-Collective-State with respect to the Global_History data structure.

Proof of Claim G.3: Immediate from Lemmas G.3.1, G.3.2, and G.3.3

Claim G.4: If there exists a set consisting of a majority of sites, each meeting Local_L1, such that all sites in the set have pre-installed the same global view and have reconciled their Global Histories, then the preconditions for completing Assign-A-Sequence will eventually be met at the leader site.

Lemma G.4.1: If there exists a set consisting of a majority of sites, each meeting Local_L1, such that all sites in the set have pre-installed the same global view and have reconciled their Global Histories, and all correct servers at the leader site have been globally constrained in this view, then either global progress is made, or a correct representative will eventually not be suspected by any correct server.

Proof: By Claim G.2, all correct servers in all sites in the

majority set preinstall the same global view. By the Leader Site timeout (T_3), if no global progress is made, then a correct server will eventually be elected at the leader site before a global view change occurs. At this point, at least $2f+1$ correct connected servers do not suspect the representative.

Lemma G.4.2: If there exists a set consisting of a majority of sites, each meeting Local_{L1}, such that all sites in the set have pre-installed the same global view and have reconciled their Global Histories, and all correct servers at the leader site have been globally constrained in this view, then either global progress is made, or a correct representative will eventually not be suspected by any correct server, and will not be suspected by a correct server before at least three local network crossings.

Proof: Follows immediately from Lemma G.4.1 and the relationship between timeouts.

Lemma G.4.3: If there exists a set consisting of a majority of sites, each meeting Local_{L1}, such that all sites in the set have pre-installed the same global view and have reconciled their Global Histories, and all correct servers at the leader site have been globally constrained in this view, then either global progress is made, or the data structures of the correct servers in the leader site will eventually be synchronized such that there are no conflicts.

Proof: By Claim G.3, none of the Prepare_OK messages received by the correct servers in the leader site cause any conflicts. By the relationship between timeouts, the leader site will eventually elect a correct representative that has the most up-to-date Prepare certificates, if any exist.

Proof of Claim G.4: Follows immediately from Lemmas G.4.1, G.4.2, and G.4.3.

Claim G.5: If there exists a set consisting of a majority of sites, each meeting Local_{L1}, such that all sites in the set have pre-installed the same global view and have reconciled their Global Histories, then eventually, if a correct representative of the leader site sends a valid Proposal message in this view, it will globally order the associated update.

Lemma G.5.1: If there exists a set consisting of a majority of sites, each meeting Local_{L1}, such that all sites in the set have pre-installed the same global view and have reconciled their Global Histories, then eventually, if a correct representative of the leader site sends a valid Proposal message in this view, the Proposal will not cause any real conflicts at any correct server in a non-leader site in the majority.

Proof: Immediate from the invariants of the Global History.

Lemma G.5.2: If there exists a set consisting of a majority of sites, each meeting Local_{L1}, such that all sites in the set have pre-installed the same global view and have reconciled their Global Histories, then eventually, if a correct representative of the leader site sends a valid Proposal message in this view, it will be received by a correct representative at all non-leader sites in the majority.

Proof: Immediate from the stability properties of the network and from relationship between T_1 , T_2 , and the Leading Site Timeout.

Lemma G.5.3: If there exists a set consisting of a majority of

sites, each meeting Local_{L1}, such that all sites in the set have pre-installed the same global view and have reconciled their Global Histories, then eventually, if a correct representative of the leader site sends a valid Proposal message in this view, the representative will receive Accept messages from at least majority-1 sites.

Proof: By Lemma G.5.1 and G.5.2, the Proposal message will not cause any real conflicts at the correct servers in the non-leader sites, and will be received by all non-leader sites in the stable majority. By the preconditions for successful completion of Threshold-Sign, and the relationship between T_1 , T_2 , and the Leading Site timeout, each non-leader site in the majority will generate an Accept message corresponding to the Proposal, which will be sent out by a correct representative before the Leading Site timeout expires. Since the sites are connected, and the leader site still has a correct representative (by T_3), the representative will receive all of the Accept messages.

Proof of Claim G.5: Immediate from Lemmas G.5.1, G.5.2, and G.5.3.

VII. PERFORMANCE EVALUATION

To evaluate the performance of our hierarchical Byzantine replication architecture, we implemented a complete prototype of our protocol including all the necessary communication and cryptographic functionality. In this paper we focus only on the networking and cryptographic aspects of our protocols, and do not consider disk writes.

A. Implementation

The OpenTC library is an implementation of RSA threshold signatures based on the ideas proposed by Shoup. There is often the need to guarantee the authenticity of messages sent by a group of individuals to another group. To protect against a representative of a site sending bogus messages, other sites must be able to verify the message authenticity. To facilitate the verification, messages originated from a site are jointly signed by a significant threshold of nodes in the site. A (n, k) threshold signature scheme can be used for the purpose and it offers several advantages. First, only a public key is needed for each site and a single verification for each message is needed for a recipient. These properties greatly reduce the overheads on key distribution and signature verification. Second, the threshold k can be set accordingly with respect to the number of malicious nodes to be tolerated. Also, the threshold k and the secret shares can be proactively changed while keeping the same public key.

Testbed and Network Setup:

We selected a network topology consisting of 5 wide area sites, assuming that there can be at most 5 Byzantine faults in each site, in order to quantify the performance of our system in a realistic scenario. This requires 16 replicated servers in each site.

Our architecture uses RSA threshold signatures [6] to represent an entire site within a single trusted message sent on the wide area network, thus trading computational power for

wide area bandwidth and latency, in the number of wide area crossings. We believe this tradeoff is realistic considering the current technology trend: end-to-end wide area bandwidth is slow to improve, while latency reduction of wide area links is limited by the speed of light.

Our experimental testbed consists of a cluster with twenty 3.2GHz Intel Xeon computers, all of them having a 64-bit architecture. On these computers, a 1024 bit RSA signature can be computed in 1.3 msec and can be verified in 0.07 msec. The leader site was deployed on 16 of the machines, and the other 4 sites were emulated by one computer each¹. The emulating computers were seen from the other sites as if they were the representatives of complete 16 server sites, for a system consisting of a total of 80 nodes spread over 5 sites. Upon receiving a packet at a non-leader site, the emulating computers were busy-waiting for the amount of time it took a 16 server site to handle that packet and reply to it, including both in-site communication and the necessary computation. The busy-waiting times for each type of packet were determined in advance by benchmarking individual protocols on a fully deployed, 16 server site. We used the Spines [12] messaging system to emulate latency and throughput constraints on the wide area links.

We compared the performance results of the above system with those obtained by BFT [3] on the same network setup with five sites, run on the same cluster, only that instead of using 16 servers in each site, for BFT we used a **total** of 16 servers across the entire network. This allows for up to 5 Byzantine failures in the entire network for BFT, instead of up to 5 Byzantine failures in each site for Steward; however, since BFT is a flat solution where there is no correlation between faults and the sites where they can occur, we believe this comparison is fair and conservative. We distributed the BFT servers such that four sites contain 3 servers each, and one site contains 4 servers.

All the write updates and read-only queries in our experiments carried a payload of 200 bytes, representing a common SQL statement.

Bandwidth Limitation:

We first investigate the benefits of the hierarchical architecture in a symmetric configuration with 5 sites, where all sites are connected to each other with 50 milliseconds latency links. A 50 millisecond delay emulates the wide area crossing of the continental US.

In the first experiment, clients inject write updates. Figure 17 shows the update throughput when increasing the number of clients, limiting the capacity of wide area links between the sites to 10, 5 and 2.5Mbps, both for Steward and BFT. The graph shows that up to 2.5Mbps, Steward is not limited by bandwidth. The system is able to process a total

¹Our implementation was tested on a complete deployment where each site is composed on multiple computers using the complete set of protocols and is currently undergoing a 5-sites DARPA red-team exercise. In order to evaluate Steward's scalability on large networks supporting many faults at each site, we used emulating computers for non-leader sites to limit the deployment to our cluster of 20 machines.

of about 84 updates/sec, being limited only by CPU, used for computing threshold signatures at the sites.

As we increase the number of clients, the BFT throughput increases at a lower slope than Steward, mainly due to the one additional wide area crossing for each update. At 10 Mbps, BFT achieves about 58 updates/sec, being limited by the available bandwidth. Similarly, at 5 Mbps it can sustain a maximum of 26 updates/sec, and at 2.5 Mbps a maximum of about 6 updates/sec. We also notice a reduction in the throughput of BFT as the number of clients increases. We believe this is due to a cascading increase of message loss, generated by the lack of a wide area flow control in the original implementation of BFT. Such a flow control was not needed as BFT was designed to work in LANs. For the same reason, we were not able to run BFT with more than 24 clients at 5 Mbps, and 15 clients at 2.5Mbps. We believe that adding a client queuing mechanism would stabilize the performance of BFT to its maximum achieved throughput, regardless of the number of clients.

The average update latency, as depicted in Figure 18, shows Steward achieving almost constant latency. The latency slightly increases with the addition of clients, reaching 190 ms when 15 clients send updates into the system. At this point, as client updates start to be queued, their latency increases linearly with the number of clients in the system. BFT exhibits a similar behavior at 10 Mbps, only that its update latency is affected by the additional number of messages sent and the additional wide area crossing, such that for 15 clients the average update latency is 336 ms. As the bandwidth decreases, the update latency increases heavily, reaching up to 600 ms at 5 Mbps and 5 seconds at 2.5 Mbps, for 15 clients.

Adding Read-only Queries: One of the benefits of our hierarchical architecture is that read-only queries can be answered locally, at each site. To demonstrate these benefits we conducted an experiment where 10 clients send mixes of read-only queries and write updates, chosen randomly at each client, with different ratios. We compared the performance of Steward and BFT when both systems are not limited by bandwidth constraints. We used links of 50 ms, 10 Mbps between the sites. Figures 19 and 20 show the average throughput and latency, respectively, of different mixes of queries and updates sent using Steward and BFT. When clients send only read queries, Steward achieves about 2.9 ms per query, with a throughput of over 3,400 queries per second. This is because all the queries are answered locally, their latency being dominated by two RSA signature operations: one at the originating client, and one at the servers answering the query.

For BFT, the latency of read-only queries is about 105 ms, and the total throughput achieved is 95 queries per second. This is expected, as read-only queries in BFT need to be answered by at least $f + 1$ servers, some of which being located across wide area links. BFT could have achieved queries locally in a site if we deployed it such that there are at least $2f + 1$ servers in each site (in order to guarantee liveness it needs $f + 1$ *correct* servers to answer queries in each site). Such a deployment, for $f = 5$ faults and 5 sites,

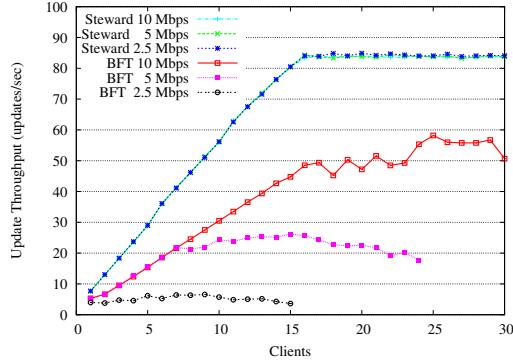


Fig. 17. Write Update Throughput

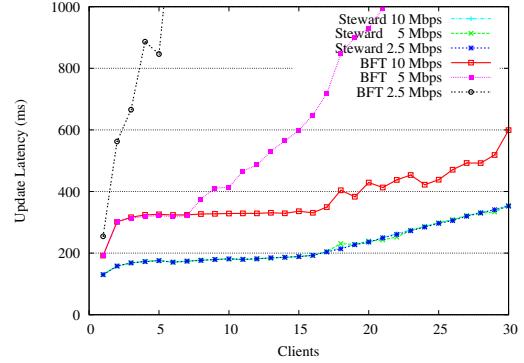


Fig. 18. Write Update Latency

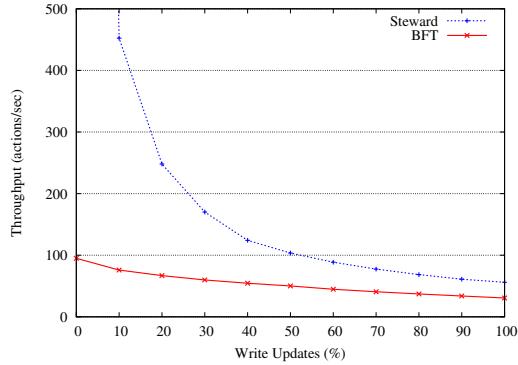


Fig. 19. Update Mix Throughput - 10 Clients

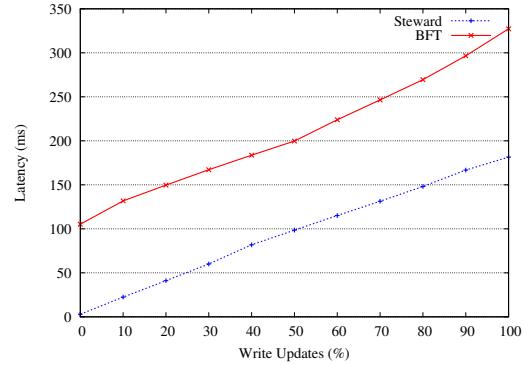


Fig. 20. Update Mix Latency - 10 Clients

would need at least 55 servers total, which will dramatically increase communication for updates, and further reduce BFT's performance.

As the percentage of write updates in the query mix increases, the average latency for both Steward and BFT increases linearly, with Steward latency being about 100 ms lower than BFT at all times. This is a substantial improvement considering the absolute value of the update latency, the ratio between the latency achieved by the two systems ranging from a factor of two, when only write updates are served, to a factor of 30, when only read queries are served. The throughput drops with the increase of update latency, such that at 100% write updates there is only about a factor of two between the throughput achieved by Steward and BFT.

Wide Area Scalability: To demonstrate the scalability of the hierarchical architecture we conducted an experiment that emulated a wide area network that covers several continents. We selected five sites on the Planetlab network [13], measured the latency and available bandwidth characteristics between every pair of sites, and emulated the network topology on our cluster in order to run Steward and BFT. We ran the experiment on our cluster, and not directly on Planetlab because Planetlab machines are not of 64-bit architecture. Moreover, Planetlab computers provide a shared environment where multiple researchers run experiments at the same time, bringing the load on almost all the machines to more than 100% at all times.

Such an environment lacks the computational power required for the two systems tested, and would artificially influence our experimental results.

The five sites we emulated in our tests are located in the US (University of Washington), Brazil (Rio Grande do Sul), Sweden (Swedish Institute of Computer Science), Korea (KAIST) and Australia (Monash University). The network latency varied between 59 ms (US - Korea) and 289 ms (Brazil - Korea). Available bandwidth varied between 405 Kbps(Brazil - Korea) and 1.3Mbps (US - Australia).

Figure 21 shows the average write update throughput as we increased the number of clients in the system, while Figure 22 shows the average update latency. As seen in Figures 21 and 22, Steward is able to achieve its maximum limit of about 84 updates/second when 27 clients inject updates into the system. The latency increases from about 200 ms for 1 client, to about 360 ms for 30 clients.

BFT is limited by the available bandwidth to a maximum of about 9 updates/sec, while the update latency starts at 631 ms for one client, and jumps to the order of seconds when more than 6 clients are introduced.

Comparison with Non-Byzantine Protocols:

Since Steward deploys a lightweight fault-tolerant protocol between the wide area sites, we expect it to achieve performance comparable to existing non-Byzantine fault-tolerant protocols commonly used in database replication systems, but with Byzantine guarantees (while paying more hardware).

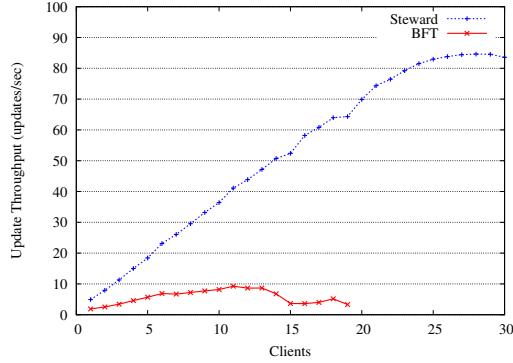


Fig. 21. Wide Area Network Emulation - Write Update Throughput

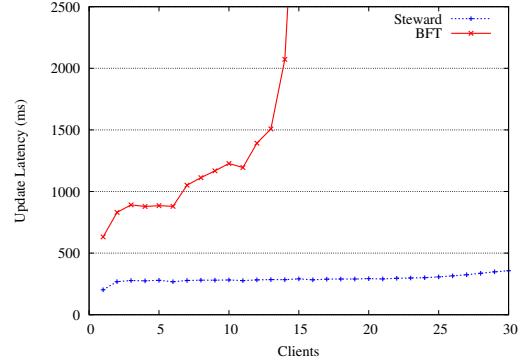


Fig. 22. Wide Area Network Emulation - Write Update Latency

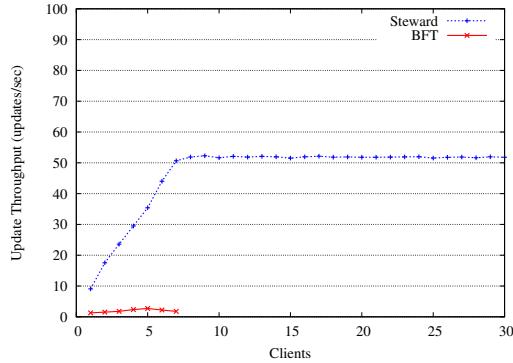


Fig. 23. CAIRN Network Emulation - Write Update Throughput

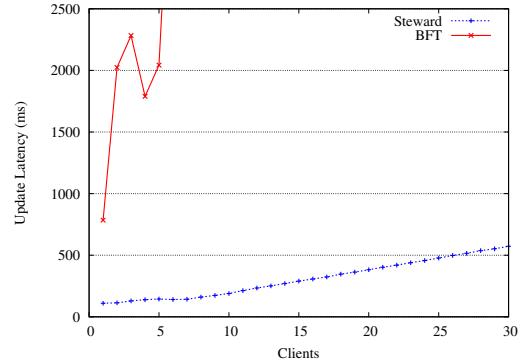


Fig. 24. CAIRN Network Emulation - Write Update Latency

In the following experiment we compare the performance of our hierarchical Byzantine architecture with that of two-phase commit protocols. In [14] we evaluated the performance of two-phase commit protocols [15] using a wide area network setup across the US, called CAIRN [16]. We emulated the topology of the CAIRN network using the Spines messaging system, and ran Steward and BFT on top of it. The main characteristic of the CAIRN topology is that East and West Coast sites were connected through a single link of 38ms and 1.86Mbps.

Figures 23 and 24 show the average throughput and latency of write updates, respectively, of Steward and BFT on the CAIRN network topology. Steward was able to achieve about 51 updates/sec in our tests, being limited mainly by the bandwidth of the link between the East and West Coasts in CAIRN. In comparison, an upper bound of two-phase commit protocols presented in [14] was able to achieve 76 updates/sec. As our architecture uses a non-Byzantine fault-tolerant protocol between the sites, it was expected to achieve comparable results with two phase commit protocols. We believe that the difference in performance is caused by the presence of additional digital signatures in the message headers of Steward, adding 128 bytes to the 200 byte payload of each update.

The high bandwidth requirement of BFT causes it to achieve a very low throughput and high latency on the CAIRN

network. The maximum throughput achieved by BFT was 2.7 updates/sec and the update latency was over a second, except when a single client injected updates in the entire system.

Summary:

The performance results we presented show that our hierarchical Byzantine architecture achieves performance comparable (though somewhat lower) to non-Byzantine protocols when run on wide area networks with multiple sites, and is able to scale to networks that span across several continents. In addition, our experiments show that the ability of our architecture to answer queries locally inside a site gives substantial performance improvements beyond the qualitative benefit of allowing read-only queries in the presence of partitions. In contrast, flat Byzantine protocols, while performing very well on local area networks, do not scale well to multiple sites across a wide area network. They have high bandwidth requirements, and use additional rounds of communication that increase individual update latency and reduce their total achievable throughput.

VIII. RELATED WORK

Agreement and Consensus:

At the core of many replication protocols is a more general problem, known as the agreement or consensus problem. There are several models that researchers considered when solving consensus, the strongest one being the Byzantine model in which a participant can behave in an arbitrary

manner. Other than the behavior of a participant (malicious or not), other relevant considerations are whether communication is asynchronous or synchronous, whether authentication is available or not, and whether the participants communicate over a flat network or not. A good overview of significant results is presented in [17]. Optimal results state that under the assumption that communication is not authenticated and nodes are directly connected, in order to tolerate f Byzantine failures, $3f + 1$ participants and $f + 1$ communication rounds are required. If authentication is available, then $f + 1$ rounds are still required, but the number of participants just has to be greater than $f + 1$ [18]. An important factor that must be taken into consideration is whether participants are directly connected or not. In [19], Dolev shows that in an arbitrary connected network, if f Byzantine faults must be tolerated and the network is $f + 1$ ($2f + 1$ if no signature exists) connected, then agreement can be achieved in $2f + 1$ rounds.

Byzantine Group Communication:

Related with our work are group communication systems resilient to Byzantine failures. The most well-known such systems are Rampart [20] and SecureRing [21]. Although these systems are extremely robust, they have a severe performance cost and require a large number of un-attacked nodes to maintain their guarantees. Both systems rely on failure detectors to determine which replicas are faulty. An attacker can exploit this to slow correct replicas or the communication between them until enough are excluded from the group.

Another intrusion-tolerant group communication system is ITUA [22], [23], [24], [25]. The ITUA system, developed by BBN and UIUC, focuses on providing intrusion tolerant group services. The approach taken considers all participants as equal and is able to tolerate up to less than a third of malicious participants.

Replication with Benign Faults: The two-phase commit (2PC) protocol [15] provides serializability in a distributed database system when transactions may span several sites. It is commonly used to synchronize transactions in a replicated database. Three-phase commit [Ske82] overcomes some of the availability problems of 2PC, paying the price of an additional communication round, and therefore, additional latency. Paxos [1] is a very robust algorithm for benign fault-tolerant replication. Paxos uses two rounds of messages in the common case to assign a total order to updates and requires $2f + 1$ replicas in order to tolerate f faults.

Quorum Systems with Byzantine Fault-Tolerance: Quorum systems obtain Byzantine fault-tolerance by applying quorum replication methods. Examples of such systems include Phalanx [26], [27] and its successor Fleet [28], [29]. Fleet provides a distributed repository for Java objects. It relies on an object replication mechanism that tolerates Byzantine failures of servers, while supporting benign clients. Although the approach is relatively scalable with the number of replica servers, it suffers from the drawbacks of flat non-hierarchical Byzantine replication solutions.

Replication with Byzantine Fault-Tolerance:

The first practical work to solve replication while with-

standing Byzantine failures is the work of Castro and Liskov [3]. Their algorithm requires $3f + 1$ replicas in order to tolerate f faults. In addition, the client has to wait for $f + 1$ identical answers (which, for liveness guarantees may require waiting for up to $2f + 1$ answers) in order to make sure that a correct answer is received. The algorithm obtains very good performance on local area networks. Yin et al. [30] propose an improvement for the Castro and Liskov approach by separating the agreement component that orders requests from the execution component that processes requests. The separation allows utilization of the same agreement component for many different replication tasks. It also reduce the number of processing storage replicas to $2f + 1$. Martin and Alvisi [31] recently introduced an algorithm that is able to achieve Byzantine consensus in only two rounds, while using $5f + 1$ servers in order to overcome f faults. This approach trades lower availability ($4f + 1$ out of $5f + 1$ connected replicas are required, instead of $2f + 1$ out of $3f + 1$ in BFT), for increased performance. The solution seems very appealing for local area networks that provide high connectivity between the replicas. We considered using it within the sites in our architecture to reduce the number of intra-site communication rounds. However, as we make use of threshold signatures inside a site, the overhead of combining larger signatures of $4f + 1$ shares would greatly overcome the benefits of using one less communication round within the site.

Alternate architectures:

An alternate hierarchical approach to scale Byzantine replication to wide area networks can be based on having a few trusted nodes that are assumed to be working under a weaker adversary model. For example, these trusted nodes may exhibit crashes and recoveries but not penetrations. A Byzantine replication algorithm in such an environment can use this knowledge in order to optimize the performance and bring it closer to the performance of a fault-tolerant, non-Byzantine solution.

Such a hybrid approach was proposed in [32], [33] by Verisimo et al, where trusted nodes were also assumed to perform synchronously, providing strong global timing guarantees. The hybrid failure model of [32] inspired the Survivable Spread [34] work, where a few trusted nodes (at least one per site) are assumed impenetrable, but are not synchronous, may crash and recover, and may experience network partitions and merges. These trusted nodes were implemented by Boeing Secure Network Server (SNS) boxes, which are limited computers designed specifically not to be penetrable.

In our opinion, both the hybrid approach proposed in [33], and the approach proposed in this paper seem viable to practically scale Byzantine replication to wide area networks. The hybrid approach makes stronger assumptions while our approach pays more hardware and computational costs. Further developing both approaches and contrasting them can be a fertile ground for future research.

IX. CONCLUSIONS AND FUTURE WORK

This paper presented a hierarchical architecture that enables efficient scaling of Byzantine replication to systems that span multiple wide area sites, each consisting of several potentially malicious replicas. The architecture reduces the message complexity on wide area updates, increasing the system's ability to scale. By confining the effect of any malicious replica to its local site, the architecture enables the use of a benign fault-tolerant algorithm over the wide area network, increasing system availability. Further increase in availability and performance is achieved by the ability to process read-only queries within a site.

We implemented Steward, a fully functional prototype that realizes our architecture, and evaluated its performance over several network topologies. The experimental results show considerable improvement over flat Byzantine replication algorithms, bringing the performance of Byzantine replication closer to existing benign fault-tolerant replication techniques over wide area networks.

X. ACKNOWLEDGMENT

This work was supported in part by grant G438-E46-2140 from The Defense Advanced Research Projects Agency.

REFERENCES

- [1] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, May 1998.
- [2] Lamport, "Paxos made simple," *SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, vol. 32, 2001.
- [3] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.
- [4] Y. G. Desmedt and Y. Frankel, "Threshold cryptosystems," in *CRYPTO '89: Proceedings on Advances in cryptology*, (New York, NY, USA), pp. 307–315, Springer-Verlag New York, Inc., 1989.
- [5] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.
- [6] V. Shoup, "Practical threshold signatures," *Lecture Notes in Computer Science*, vol. 1807, pp. 207–223, 2000.
- [7] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *CRYPTO '95: Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, (London, UK), pp. 339–352, Springer-Verlag, 1995.
- [8] L. Zhou, F. Schneider, and R. van Renesse, "APSS: Proactive Secret Sharing in Asynchronous Systems."
- [9] P. Feldman, "A Practical Scheme for Non-Interactive Verifiable Secret Sharing," in *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, (Los Angeles, CA, USA), pp. 427–437, IEEE Computer Society, IEEE, October 1987.
- [10] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust threshold dss signatures," *Inf. Comput.*, vol. 164, no. 1, pp. 54–84, 2001.
- [11] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.
- [12] "The spines project, <http://www.spines.org/>."
- [13] "Planetlab." <http://www.planet-lab.org/>.
- [14] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu, "On the performance of consistent wide-area database replication," Tech. Rep. CNDS-2003-3, December 2003.
- [15] K. Eswaran, J. Gray, R. Lorie, and I. Taiger, "The notions of consistency and predicate locks in a database system," *Communication of the ACM*, vol. 19, no. 11, pp. 624–633, 1976.
- [16] "The CAIRN Network." <http://www.isi.edu/div7/CAIRN/>.
- [17] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *Fundamentals of Computation Theory*, pp. 127–140, 1983.
- [18] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal of Computing*, vol. 12, no. 4, pp. 656–666, 1983.
- [19] D. Dolev, "The byzantine generals strike again," *Journal of Algorithms*, vol. 3, no. 1, pp. 14–30, 1982.
- [20] M. K. Reiter, "The Rampart Toolkit for building high-integrity services," in *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, (London, UK), pp. 99–110, Springer-Verlag, 1995.
- [21] K. P. Kuhlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing protocols for securing group communication," in *Proceedings of the IEEE 31st Hawaii International Conference on System Sciences*, vol. 3, (Kona, Hawaii), pp. 317–326, January 1998.
- [22] M. Cukier, T. Courtney, J. Lyons, H. V. Ramasamy, W. H. Sanders, M. Seri, M. Atighetchi, P. Rubel, C. Jones, F. Webber, P. Pal, R. Watro, and J. Gossett, "Providing intrusion tolerance with itua," in *Supplement of the 2002 International Conference on Dependable Systems and Networks*, June 2002.
- [23] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders, "Quantifying the cost of providing intrusion tolerance in group communication systems," in *The 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, June 2002.
- [24] P. Pandey, "Reliable delivery and ordering mechanisms for an intrusion-tolerant group communication system." Masters Thesis, University of Illinois at Urbana-Champaign, 2001.
- [25] H. V. Ramasamy, "A group membership protocol for an intrusion-tolerant group communication system." Masters Thesis, University of Illinois at Urbana-Champaign, 2002.
- [26] D. Malkhi and M. K. Reiter, "Secure and scalable replication in phalanx," in *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, (Washington, DC, USA), p. 51, IEEE Computer Society, 1998.
- [27] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Journal of Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.
- [28] D. Malkhi and M. Reiter, "An architecture for survivable coordination in large distributed systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 187–202, 2000.
- [29] D. Malkhi, M. Reiter, D. Tulone, and E. Ziskind, "Persistent objects in the fleet system," in *The 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, (2001), June 2001.
- [30] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault-tolerant services," in *SOSP*, 2003.
- [31] J.-P. Martin and L. Alvisi, "Fast byzantine consensus..," in *DSN*, pp. 402–411, 2005.
- [32] M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo, "Efficient byzantine-resilient reliable multicast on a hybrid failure model," in *Proc. of the 21st Symposium on Reliable Distributed Systems*, (Suita, Japan), Oct. 2002.
- [33] P. Veríssimo, "Uncertainty and predictability: Can they be reconciled," in *Future Directions in Distributed Computing*, no. 2584 in LNCS, Springer-Verlag, 2003.
- [34] "Survivable spread: Algorithms and assurance argument," Tech. Rep. Technical Information Report Number D950-10757-1, The Boeing Company, July 2003.

Appendix B

- “Enhancing Distributed Systems with Mechanisms to Cope with Malicious Clients” Y. Amir, C. Danilov, J. Lane, M. Miskin-Amir and C. Nita-Rotaru. Technical Report CNDS-2005-4, Distributed Systems and Networks lab, Johns Hopkins University, www.dsn.jhu.edu.

Enhancing Distributed Systems with Mechanisms to Cope with Malicious Clients

Technical Report CNDS-2005-4 - December 2005
<http://www.dsn.jhu.edu>

Yair Amir, Claudiu Danilov, John Lane
Johns Hopkins University
{yairamir, claudiu, johnlane}@cs.jhu.edu

Michal Miskin-Amir
Spread Concepts LLC
michal@spreadconcepts.com

Cristina Nita-Rotaru
Purdue University
crisn@cs.purdue.edu

Abstract— In this paper we identify a major security vulnerability in distributed systems: compromised clients under adversarial control can use the system within their authorized access rights and authenticated channels to deliberately insert incorrect data. A significant problem is that when a malicious client insider is discovered, it is hard to quickly assess the scope of the damage, and identify corrupt and suspected updates.

We propose Accountability Graph, a mechanism that can assist applications in coping and recovering from such attacks. The tool provides accountability enforcement and causality tracking of updates and their dependencies. Upon detection of incorrect data (e.g. by an external intrusion detection mechanism or human assessment), the Accountability Graph will quickly classify all updates in the system as either corrupted, suspected or not affected. The practicality and usefulness of the approach is demonstrated based on the requirements of three different applications: an open source software development project, a military common operation picture application, and a national emergency response system. The Accountability Graph can also be used for risk assessment and vulnerability analysis with respect to the above attack.

I. INTRODUCTION

Many distributed services are implemented following a model where a set of servers replicate the service and coordinate their actions to answer client requests while maintaining the consistency of the data. The most basic operations performed by clients are querying the servers or updating data maintained by the servers. Security is a major concern for such systems that often operate over unsecure networks such as the Internet. Significant work conducted in the last several years to develop mechanisms for Byzantine replication [1], [2], [3], access control [4], [5], [6] and intrusion detection [7], [8], [9] provides the support for designing secure distributed services. Specifically, the servers and their operating system are protected against intrusions, corrupted servers are tolerated by running Byzantine replication algorithms, access to resources is tightly enforced by using access control mechanisms, while client actions are monitored by intrusion detection systems.

Although such systems may seem difficult to attack, they overlook that the weakest link is represented by the clients (often communicating with the servers over wireless channels) and the most critical asset is the data itself. Thus, very harmful attacks can come from compromised clients, targeting the data

correctness: One or more compromised clients can *use* the system within their authorized parameters to create or inject incorrect inputs or updates to some servers. The (Byzantine) replication algorithms will propagate this information among all servers, corrupting the state of the system so that it will no longer reflect reality. Several observations are important. First, the Byzantine replication protocols running on the servers will replicate data already compromised, so they will not be able to address the attack. Second, these incorrect updates may not be detected immediately, impacting other clients subsequently querying the system and basing their decisions on the erroneous state. This creates a cascading effect in which further created updates are also erroneous because they are based on malicious data. Third, although intrusion detection mechanisms deployed in the system may eventually detect the compromised clients, assessing the extent of the damage and identifying the other components of the system that were affected, or are suspect and need further investigation is very challenging and is not provided by the mechanism mentioned above.

The effect of such an attack can be devastating for applications that are highly dependent on the correctness of their data. For example, in collaborative open-source software development (e.g. Linux), multiple individuals create or augment existing source code. The inherent interdependency between software packages enables a malicious update to one package to significantly impact other components of the system. It is important to identify the packages that may be affected by corrupt code injected into the system, and determine the risk and vulnerabilities associated with it.

Other examples are command and control information systems, such as those used by the military [10] or by emergency response personnel [11]. In such systems, users update the state of the operational situation and make decisions based on it. Correctness of the data is critical, and any misleading information can result in loss of life. A malicious insider can inject authorized yet incorrect information that may mislead honest users and cause them, in turn, to make additional erroneous updates.

Our Focus and Contribution: A major problem with secure distributed systems is that when a malicious client

insider is discovered, it is hard to quickly assess the scope of the damage, and identify corrupt and suspected updates. Therefore, the system is not able to regenerate and recover to a clean state without the effects of these updates. Based on our experience building secure reliable systems, we make the observation that in the best case, this is considered an application-specific issue, and the system infrastructure provides no support in addressing it. Most of the time, this problem is not considered at all. The goal of this work is to raise awareness to this important problem and to show how the distributed infrastructure can assist the application in recovering from such attacks. Our preliminary results based on the requirements of three different applications demonstrate the practicality of our solution.

We propose Accountability Graph, a generic mechanism that provides accountability enforcement and causality tracking of updates and their dependencies in a directed acyclic graph with periodic snapshots. Upon detection of incorrect data, the system traces the data to the corrupt update that generated it, and from that, the Accountability Graph enables us to mark all causally dependent updates as corrupted or suspected. We mark all subsequent updates made by the malicious client that generated the corrupted update as corrupt, and use a standard graph traversal to identify and mark as suspicious all other updates that recursively depend on corrupted updates. No less important, the system is assured that all unmarked updates are not affected by the discovered incorrect data. Our proposed solution can use any intrusion detection mechanism (or human input) that will provide the initial detection. One or several servers forming the underlying distributed service can decide to maintain the graph, the coordination between the servers, including the ordering of the updates, will ensure that the graph looks the same at each server. There is no central authority or point of failure, any server can decide at any time if it will build the graph for events happening in the system.

The contributions of the paper are:

- We identify a significant attack against distributed services mounted by malicious clients that deliberately insert incorrect data through authorized channels.
- We propose a generic mechanism, Accountability Graph, that tracks the dependencies between all of the updates in the system. When notified about compromised clients or corrupt updates by external mechanisms (such as intrusion detection, human assessment, application-specific knowledge), the Accountability Graph can classify data as corrupt, suspect, or not affected.
- We demonstrate the usefulness of our solution in three different applications: an open-source software development project, a military common operation picture application, and a national emergency response system. We show that the overhead associated with our solution is reasonable in these cases.
- We present an additional benefit of the Accountability Graph, namely the ability to conduct risk assessment and vulnerability analysis with respect to the compromised client attack.

The rest of the paper is organized as follows. We present a description of the model considered in this paper in Section II. Section III presents a detailed description of the Accountability Graph. We demonstrate the usefulness and feasibility of our approach for several applications in Section IV. In Section V we present the performance of the Accountability Graph, and in Section VI we survey related work. We conclude the paper in Section VII.

II. SYSTEM MODEL

We assume a general message-passing system where one or more servers respond to requests from clients. The requests submitted by clients can be updates (write operations), or queries (read operations). Communication is asynchronous.

We assume that each client has a public and private key pair. Servers know the public keys of all clients that connect to them. Cryptographic techniques such as public-key digital signatures, message authentication codes, and message digests produced by collision-resistant hash functions, are used to provide non-repudiation, message integrity and authentication of messages. We assume that the adversary is computationally bounded such that he cannot subvert these cryptographic techniques.

Clients communicate with the servers using secure channels: the communication is protected from an external adversary by using encryption, all messages are authenticated and carry integrity information which prevents an external adversary from injecting or modifying packets.

The adversary can compromise any number of clients, coordinate the attack, delay communication, modify, delete or replay a message, or simply generate and deliberately send incorrect data. The adversary cannot delay indefinitely correct clients. When a client node is compromised, the adversary has full control over that node, including access to all cryptographic keys stored on the machine.

We assume that there are external mechanisms that can detect that clients were compromised or that they submitted updates containing incorrect data. This can be done by employing tools such as intrusion detection systems, or by having a human review the data offline. The intrusion detection is not instantaneous, i.e. clients can inject several malicious updates before they are detected. Correct clients may use affected data in their decisions (and therefore create incorrect updates themselves) before a malicious update is detected. Thus, the damage caused by a malicious update can affect future updates (not necessarily made by the malicious client), as well as queries that will propagate incorrect information to honest clients.

III. ACCOUNTABILITY GRAPH: DESIGN, IMPLEMENTATION AND EXPRESSIVENESS

Based on the observations formulated in Section I, we believe that there is a need for mechanisms that provide the following:

- Corrupted data isolation: when notified that an update is incorrect, the system can identify the data affected by it,

and provide fast feedback about all other compromised updates.

- Automatic regeneration of non-corrupted states: allow automatic regeneration of a view of the data that includes non-contaminated data, based on initial information about problems that occurred and building knowledge about the extent of the problem.

We address these requirements by building an Accountability Graph, a directed acyclic graph that maintains causal dependency of updates, allowing data classification in the light of a compromised client or incorrect update, and facilitating automatic regeneration of a correct state. We note that the Accountability Graph provides a useful tool to help assess risk assuming that specific participants were compromised at known times, or that specific updates were incorrect. Below we describe our design, with focus on the construction and traversal of the causality graph.

A. Design Overview

To track client updates, we construct the causality graph as follows. Every update in the system is uniquely identified, includes the ID of its client creator, and is signed by that client. Every update also contains the identifier and digital signature [12], [13] of every previous update directly responsible for data on which the new update depends. In addition, we assume a standard causal relationship where every update depends on the previous update created by the same client. The Accountability Graph is maintained such that every update is a node in the graph and there is a directed link from that update to all the updates on which it depends. The dependency information is usually specific to the application. In some cases, dependencies may be introduced by clients themselves. In other cases, dependencies occur based on the flow of the application, while in the most conservative case we may consider that an update introduced by a client depends on all the updates previously reported to that client. In this work we do not make any assumption about the nature of dependencies.

When data is detected as incorrect, it is traced to the corresponding update. Then, by traversing the graph, corrupted and suspected updates are marked. For example, in Figure 1(a), an update of client C_4 is found to be incorrect, and the generating client, C_4 , is presumed malicious. Subsequent updates from that client are marked as corrupt and all the updates that depend on them are marked as suspicious, as shown in Figure 1(b). Notice that the arrows indicate dependency relations (i.e. if update A depends on update B , there is an arrow from A to B). Therefore, the edges in the graph are traversed in the opposite direction of the arrows. The Accountability Graph can now present various views of the system state based on these markings, and the system can regenerate its state as needed based on the updates that are deemed valid. The system can consider only the clean updates, or it can consider both clean and suspected updates.

To limit the memory required for storing the causality graph and the processing required for state regeneration, the system can use periodic posteriori snapshots. Every epoch,

for example 12 hours, a snapshot of the system state as of 12 hours ago is calculated and stored. This limits the processing required for state regeneration when bad data is discovered, as calculations are performed from the last valid snapshot (usually the last one). Of course, the length of the actual epoch depends on the rate of the updates in the system.

The pseudocode for the operations used to create and traverse the graph is presented in Algorithm 1. The *SubmitUpdate* function creates a node containing the client's identity and a unique sequence number. It also places the node in an associative container so that it can be found using its identifier. The *AddDependencies* function adds directed edges from all of the nodes in a specified list to the node specified by its identifier. Finally, the function *GetSuspectedUpdates* performs a standard graph traversal, beginning at the specified corrupt node, and marks the other nodes as corrupt, suspect, or not affected.

B. Optimization of the Accountability Graph

In the algorithm described above, when a client submits an update, the Accountability Graph automatically creates an edge connecting the update being added to the previous update submitted by that client. In Figures 1(a) and 1(b), these automatically generated edges correspond to the vertical arrows. Note that FIFO edges are always present in standard network-level causality graphs [14]; they are a conservative approximation to true, application-level, causality. These edges are also vital for our Accountability Graph. When a corrupt update is discovered, we assume that the client that generated this update is malicious and that all subsequent updates submitted by the malicious client are also corrupt. A single traversal beginning at the corrupt update can mark every corrupt and suspected update precisely because of these vertical lines. The FIFO edges link all of the updates that we assume are corrupt. Note that the malicious client cannot alter or remove these edges because it is not responsible for generating them.

The algorithm we have specified is simple and computationally efficient, but it suffers from an important problem. Consider what might happen if an honest client's update does not depend on its previous update. For example, in a military Common Operations Picture, a status update, U_s , made by a tank about its fuel, ammunition, and position is actually independent of the last update submitted by this tank. Suppose that a prior update submitted by the tank was dependent on an update made by a malicious client. Then, U_s would be marked as suspect during a graph traversal. In addition, anything with recursive dependencies on U_s would be labeled as suspicious. The result is a high false positive rate.

Before presenting our solution to this problem, we make one important observation: If an update of an honest client is compromised, future updates introduced by that client can be trusted unless they depend on corrupted data themselves. Our modified algorithm automatically adds FIFO edges as described above. However, these edges do not necessarily need to be used. By default, we disable the FIFO edges in the directed acyclic graph so that the graph traversal will not

Algorithm 1 Accountability Graph Operations

Each node (update) maintains:

Node

- Int id // A unique id for this node
- Int client_id // A unique client id
- Node dependents[] // A list of those nodes that are dependent on this node
- Int classification // CORRUPT, SUSPECT, or NOT_AFFECTED

Global Variables:

- Int next_id = 0 //this is used to create the id for the Node
- Int suspects[] //a list to store the ids of suspect nodes
- A Map of all nodes, indexed by the node's id

Int SubmitUpdate(Int client_id)

- next_id = next_id + 1
- let n = new Node with id next_id
- add n to the A-DAG //store the node in a container where it can be accessed via its id
- add the last update of this client to n.dependents
- return next_id

AddDependencies(Int dependent, Int []dependencies)

- let n = node having id of dependent
- for each i in dependencies
 - let n_d = node having id of i
 - add n to n_d.dependents //this creates a directed edge from n_d to d

Int[] GetSuspectedUpdates(Int corrupt_node)

- clear suspects //remove all suspect updates
- set the classification to NOT_AFFECTED for all nodes
- let next = 0
- let n = Node with id equal to corrupt_node
- add n to suspects
- set n.classification = CORRUPT
- while suspects.size() > next
 - let p = Node with id equal to suspects[next]
 - next = next + 1
 - for each c in p.dependents
 - if c.id is not in suspects
 - add c.id to suspects
 - if c.client_id == n.client_id, set c.classification = CORRUPT
 - else, set c.classification = SUSPECT
- return suspects //return the suspects

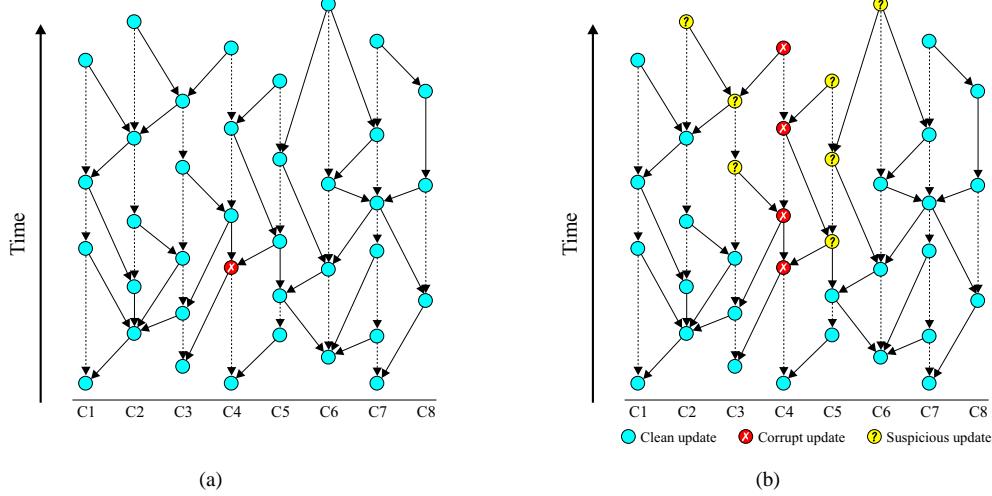


Fig. 1. Accountability Graph Example

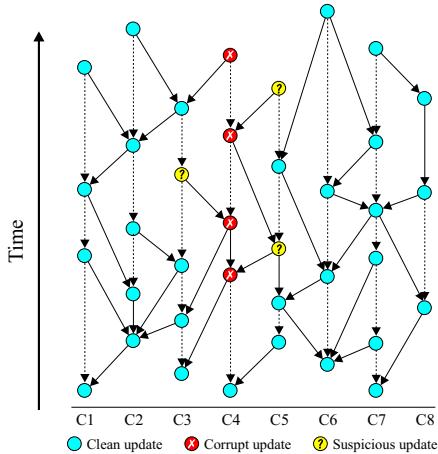


Fig. 2. Removing FIFO Edges

use them. When a corrupt update is found, we first enable only those FIFO edges that connect updates made by the malicious client (i.e. the client that generated the corrupt update). Then, a traversal is done as previously described. Figure 2 shows the same scenario as 1(b) with this optimization; now, fewer updates are labeled as suspect. Note that the improved Accountability Graph can express when a client submits an update that really does depend on its previous update. As in the original algorithm, malicious clients are unable to alter these edges because they do not generate them. We believe that this change is important not only because it reduces false positives, but also because it draws attention to the differences between conservative, network-level causality graphs and pruned, application-level causality graphs. By working at the application-level and allowing clients and/or other dependency sources to specify dependency relations, we improve the accuracy and utility of the Accountability Graph.

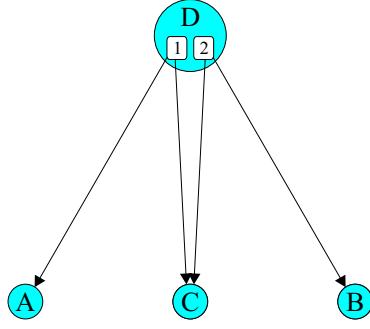


Fig. 3. OR Dependency

C. Enriching the expressiveness of the Accountability Graph

The Accountability Graph construction algorithm described above has limited expressiveness; the integrity of an update depends on the integrity of all of its dependencies. If C depends on the set $\{A, B\}$, then C is suspect if A or B is suspect. This means that the integrity of C depends on A AND B . Expressing dependencies with only *AND* operators does not always adequately capture a dependency relation. For example, consider an application where two motion sensors, S_a and S_b , cover the same area. Suppose that both sensors make an update stating that there is motion in this area. An administrator receives these two updates and makes an update, D , that dispatches a security guard to the area. The integrity of D depends on the integrity of S_a OR S_b . The administrator is conservative and therefore would have dispatched the guard even if only one of the sensors reported motion. The original algorithm cannot express this dependency.

We increase the expressive power and, thereby, the accuracy of our dependency graph by introducing an *OR* operator. The

basic idea is shown in Figure 3, which depicts that D depends on $(A \text{ OR } B) \text{ AND } C$. Note that node D contains two numbered cells and that dependency arrows originate from these cells. During a graph traversal, both numbered cells in D must be visited before D is marked as suspect. If only A is suspect, then only cell 1 will be visited. Similarly, if only B is suspect, then only cell 2 will be visited. However, if C is suspect, then both cell 1 and 2 will be visited. In general, we can modify our algorithm so that it can express all dependency relations written using linear combinations of OR and AND operations.

The OR operator enables the Accountability Graph to express dependencies on redundant sources of information. In the above example, the motion sensors provide the same information. If a client bases an update on both sensors, the update remains unaffected even if one of the sensors was corrupt. This is important because fault tolerance is commonly improved by using redundancy. Therefore, the OR operator increases the usefulness of the Accountability Graph as an offline analysis tool. When the Accountability Graph is used to assess system vulnerabilities, the OR operator can be used to show the benefits of strategically placed redundancy.

IV. CASE STUDIES

To demonstrate the feasibility of the Accountability Graph for real applications, we consider three applications for which we believe our mechanisms can offer great benefits. These applications are drawn from: open-source software projects, network-centric warfare applications, and information access to national emergency systems.

A. Collaborative Open-Source Software Projects

Many applications today rely on open source software projects, such as Debian [15], Red Hat [16], Apache [17], and Gnome [18]. Such projects are collaborative, distributed over several machines, and involve many participants. For example, the Debian project has over 1000 registered developers managing about 10000 software packages supporting 12 different platforms. If a critical machine is compromised, all packages that passed through it during creation are suspect. If a package is compromised, any other packages that use it are compromised. If a client is untrustworthy, all packages that client was involved in are suspect. Unfortunately such incidents are a reality: in 2001 [19], the public server used by the Apache Software Foundation to provide the source code repository, binary distribution, web services, and public mailing lists was compromised; in 2001 a developer introduced a Trojan horse in one of the Debian packages, while more recently in 2003 Debian was again in the news [20] when four servers were compromised, one of them hosting security updates; Gnome also was the target of an attack in 2004 [21].

We chose to use Red Hat software packages (RPMs) as a representative open source software project because RPMs are widely used and because the Red Hat Package Manager contains tools for extracting dependency information. Each RPM package contains a set of capabilities such as programs,

libraries and data, and may require other capabilities to already be installed on the system. The fifteen Red Hat distributions using RPMs contain 52,984 software capabilities, spanning 7 years of development. The distributions we considered, the number of RPMs, and the number of software capabilities in each distribution are presented in Table I.

Version	Number of RPMs	Number of Capabilities
RedHat 4.2	458	632
RedHat 5.0	482	693
RedHat 5.1	523	789
RedHat 5.2	573	891
RedHat 6.0	645	1523
RedHat 6.1	718	1691
RedHat 6.2	743	2049
RedHat 7.0	865	2113
RedHat 7.1	1016	2918
RedHat 7.2	1231	3862
RedHat 7.3	1438	5715
RedHat 8.0	1472	6432
RedHat 9	1402	7128
RedHat Fedora 1	1466	7754
RedHat Fedora 2	1619	8804
TOTAL	14651	52984

TABLE I
RED HAT DISTRIBUTIONS

Each package provides one or more software capabilities such as programs, libraries, or data. Some capabilities have many thousands of directly or recursively dependent capabilities resulting in very complex dependency relationships. If one of the capabilities is corrupt, it can potentially affect all of the capabilities that depend on it. It is very difficult and time consuming to manually analyze such a system and determine the set of capabilities that may be affected by a corrupt capability. The Accountability Graph automates this task and thus can be extremely useful when corrupt software is discovered and distribution administrators want to promptly determine the extent of the possible damage.

The integrity of a software capability in some specific distribution depends on the integrity of the same software capability in all older distributions. From one distribution to the next, capabilities evolve slowly and source code added to one version is generally present in many subsequent versions. Suppose that a malicious programmer added a vulnerability to the source code of the encryption capability `libcrypt.so.1` in RedHat 5.0. Clearly, anyone using a capability that directly or recursively depended on this version of `libcrypt.so.1` could have been affected. It is also possible that the malicious code has propagated to subsequent versions of `libcrypt.so.1` in RedHat 5.1 through Fedora 2. Therefore, when a compromise of `libcrypt.so.1` in RedHat 5.0 is found, it is important to obtain a list of all software packages that depend on this version or on any subsequent version. Note that because `libcrypt.so.1` is a shared library, fixing all versions of `libcrypt.so.1` will produce a properly functioning system assuming that static linking was not used. However, we are also concerned with finding any capability that may have been vulnerable during the time

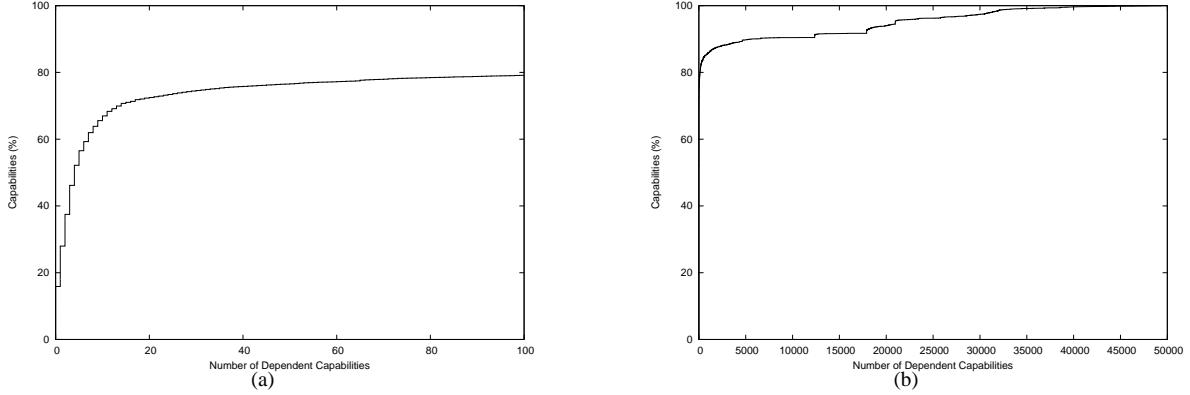


Fig. 4. CDFs of the percentage of capabilities (on the Y axis) having less than the specified number of dependent capabilities (on the X axis).

that the encryption library was corrupt. In this example, all capabilities that depend on `libcrypt.so.1` may have stored data that is insecure and therefore should be examined.

Below, we analyze the dependency graph generated based on the RPM packages. Each RPM provides a set of software capabilities. Commonly, the provided set contains only one capability. Other RPMs provide several capabilities comprised of applications, shared libraries, and other software related files. An RPM may also require a set of capabilities. The capabilities that are provided depend on the capabilities that are required. Using these sets, we created a mapping from each capability to its dependencies. More precisely, in the Accountability Graph, software capabilities represent nodes, edges are drawn between the provided capabilities and those that they require, and, even though not specified in the RPM system, developers who create software packages are considered clients. For each distribution, we constructed a list of tuples having the form: $(C_n, \text{dependencies of } C_n)$. Then, we added all of these tuples to one graph. Within each distribution, there are many dependencies and some of these reflect recursive relationships. A distribution sometimes included more than one version of a capability. When this occurred, we determined a causal order based on version numbering and made a dependency chain so that the newest version recursively depended on all prior versions. We linked the oldest version of a capability in Distribution n to the newest version in Distribution n-1.

After constructing the dependency graph, we ran a traversal beginning at each software capability and retrieved a list of all dependent capabilities. Figures 4(a) and 4(b) show cumulative distribution function (CDF) plots of the percentage of capabilities (on the Y axis) having less than the specified number of dependent capabilities (on the X axis). The figures differ only in the range shown on the y axis. We see that about 65% of capabilities have less than 20 other capabilities that depend on them. However, 20% have over 100 dependent capabilities, and furthermore, about 10% of the capabilities have over 15000 dependent capabilities. Capabilities having the greatest number of dependents include: “`libc.so.5`” (48288 dependents), “`filesystem = 1.3-1`” (48901 dependents), “`setup`

= 1.7-2” (48915 dependents) all from RedHat 4.2, and “`/sbin/ldconfig`” (49441 dependents) from RedHat 5.0.

The RPM dependency graph described above shows that, generally, a corrupt capability affects a relatively small number of other capabilities. However, some capabilities can affect many others. Therefore, the Accountability Graph can be used to identify the capabilities that would cause the most damage if they were corrupted; these capabilities represent system-wide vulnerabilities. Also note that the complex recursive dependency relations and sometimes large number of dependents make it very difficult to manually identify suspect capabilities when a corrupt capability is discovered. We believe that this case study illustrates the usefulness of the Accountability Graph both as an offline analysis tool and as an online damage assessment tool.

B. A Military Common Operation Picture Application

The military Common Operation Picture (COP) application provides a current view of the battle space shared by all friendly forces, and enables planning and coordination of the forces. The information provided by the COP may include the location of friendly and enemy units, current level of available supplies and ammunition in each unit, location of natural and man made obstacles, currently executed tactical plans and future possible plans for the different units, etc. Authentication and access control strictly determines who is allowed to view or update different parts of the operational picture. Participants constantly monitor the COP, modify their plans, and issue commands as the situation progresses.

Such applications depend heavily on the fact that information provided to the system is correct. The following scenario illustrates this problem: An update, coming from a compromised intelligence officer computer, that updates the location of an enemy unit to be 3 km south of where it actually is. This update will be accepted by the system because the intelligence officer is authenticated and is authorized to make it. A logistics officer that needs to re-supply a friendly unit, notices the location of the enemy unit according to the COP, and plots a path that will avoid the enemy. This path is also updated into the COP. The unit that is being supplied selects a

location to meet with the logistics convoy based on the updated path. In parallel, a friendly commando unit plans to attack the enemy unit. Once it is discovered that the enemy unit location is incorrect, the Accountability Graph can quickly mark the plans of the logistics convoy, the supplied unit and the commando unit. These plans were dependent directly or indirectly on the incorrect update and will have to be re-evaluated.

In this application, the clients are all the participants authorized to update any part of the common operation picture state. The nodes of the Accountability Graph are the updates to the state, and the edges of the graph refer to the past updates that influenced the decision to make the dependent update.

The Common Operation Picture application is not large compared with current hardware capabilities, and allows storing all the updates throughout the duration of a military engagement (a few months time). To explore the feasibility of the Accountability Graph to support this kind of application with adequate performance, we need to estimate the size of the graph. If we consider tracking about 5000 units, each causing the generation of about one update per minute, then over 12 hours this scenario generates about 3,600,000 updates. A snapshot of the COP state, taken every 12 hours, limits the required calculation for traversing the Accountability Graph to this number of nodes. Our experiments provided in Section V indicate that the Accountability Graph can provide an answer in matter of seconds for a dependency graph of this size.

C. Information Access for National Emergency Response Systems

The Clinicians' Biodefense Network (CBN) [22], [23] is a nationwide Internet-based information exchange system designed specifically for use by US-based clinicians in the aftermath of a bioterrorist attack. The network is managed and operated by the Center for Biosecurity of the University of Pittsburgh Medical Center. CBN was designed to facilitate communication and timely exchange of accurate and precise information among clinicians in the event of bioterrorism, and provide practitioners around the country with clinically oriented information quickly enough to guide decision-making

The design of the CBN envisioned communication between the network editorial staff, network contributors, and many thousands of network subscribers. The network data contributors are highly trusted clinicians and prominent experts who provide critical clinical information to the network, and comment on information provided by other data contributors. Several hundred clinical experts and leaders were expected to participate as data providers. The number of information updates during an emergency situation could reach several thousand per day.

The Clinicians' Biodefense Network architecture was designed to employ state of the art security mechanisms. Clinical leaders, experts, and network administrators need specific credentials in order to provide information to the network.

The network must provide accurate, correct and timely information. The potential exists for malicious users who

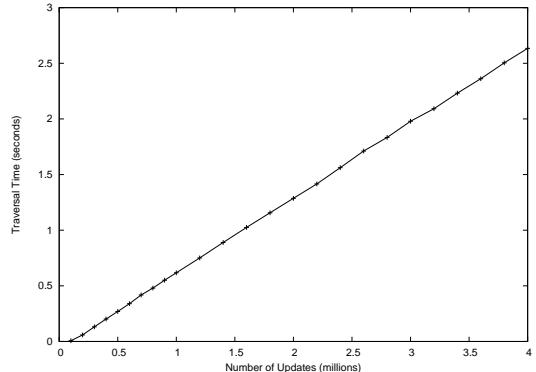


Fig. 5. Traversal time as a function of the number of updates.

impersonate a network content contributor, e.g. by stealing a password or finding other ways to infiltrate the network, to provide misleading data that could be dangerous and potentially life-threatening. For example, during a Biodefense attack, an update can incorrectly report identified cases in the wrong locations, in order to create confusion and hamper the response to legitimate cases. Further updates by other data providers can be based on this misleading data, and must also be identified and reevaluated once the malicious update is detected.

In this application, the clients are the clinical leaders and experts who provide data to the network. The nodes of the accountability graph are the messages sent on the network. The edges of the graph refer to the past messages that this message is in response to.

The type of scenarios described in the Common Operation Picture operation exist here, although the size of the CBN state and overall number of nodes in the graph is considerably smaller.

V. PERFORMANCE OF THE ACCOUNTABILITY GRAPH

The Accountability Graph keeps track of all updates introduced into the system and their dependency on other updates. It may appear that this mechanism introduces an unacceptably high overhead due to storing and processing the updates. The goal of this section is to analyze the resulted overhead and to show that for many applications it does not affect the performance significantly.

Intuitively, there are several factors that can affect the time it takes to build or traverse the graph: the number of updates (nodes) in the graph, the number of clients, the depth of the graph, and the number of dependencies (edges) in the graph. We implemented a simple data structure in C++ using the STL library and conducted several experiments to evaluate its performance as a function of these factors.

Experiment Set-up: In the experiments presented below we make no assumption about the end application, and consider a random graph with the following structure: The Accountability Graph is constructed based on time-slices in which updates are committed. Each update has a corresponding node in the graph. The nodes are organized in a two

dimensional matrix having a column for each client and a row for each time slice. The graph was built and traversed on a Intel Pentium IV 2.8GHz computer with 1GB RAM.

To initialize the graph, all clients submit one update at time t_0 . These updates are not dependent on any other update. At every subsequent time-slice, all clients submit one more update. Thus, at t_1 and after, a client's update depends on a constant number of nodes in prior time-slices. This defines the number of dependencies of that particular update.

In the experiments below we constrain the number of rows on which an update can depend. This is because in practice, only a snapshot of the dependents will be preserved. Note that the number of rows depended upon is an upper bound. For example, if this number is 20, an update submitted in time-slice t_{100} can depend on any update submitted between t_{80} and t_{99} , inclusive. Dependencies are selected randomly within the defined range of time-slices with uniform probability.

Number of updates: Figure 5 shows traversal time as a function of the number of updates in the graph. The graph traversal starts at the first time-slice, on the first update of the client having an identifier equaling $N/2$, where N is the number of clients. For this experiment we fixed the number of clients to 20, the number of dependencies to 5 and the number of prior time-slices in which nodes can have dependencies to 20. We observe that the traversal time grows linearly with the number of updates, and that for about 4 million updates, the traversal time takes less than 3 seconds. We believe that several seconds response time for marking potentially affected updates is reasonably fast to serve current applications. For example, a distributed replicated system that can handle around 80 updates per second, with a snapshot taken every 12 hours will accumulate about 3.5 million updates between snapshots. Building a 4 million dependency graph as described above took less than 20 seconds, and the data structure occupied about 225 MB of memory.

Number of clients: Figure 6 shows the traversal time and the number of nodes traversed, as a function of the number of clients submitting updates, in a scenario where the number of nodes is 4 million, and there are 5 dependencies per node. New updates can depend on the updates in previous 100 time-slices. The number of nodes visited decreases as the number of clients increases. Note that because the number of updates is fixed, the number of time-slices decreases as the number of clients increases. One node is visited in time slice t_0 . Approximately 5 nodes in t_1 are dependent on this node. In t_2 , approximately 5^2 nodes are dependent, recursively on the original node in t_0 . This trend continues (at a decreasing rate because of overlapping dependents) until all nodes in a time-slice are marked as suspect. As the number of clients increases, it takes a larger number of time-slices before all clients in each subsequent time-slice are marked as suspect, and therefore, fewer nodes are traversed as the number of clients increases.

The traversal time is initially large because of CPU cache misses, since dependent nodes are spread across a range of memory proportional to the number of clients. It then increases slightly as the number of clients increases to 10,000 due to

an increase in cache misses. Then it decreases because the number of nodes traversed decreases.

Number of dependencies: Figure 7 shows the traversal time and the number of nodes traversed as a function of the number of dependencies, when the number of nodes, clients, and dependency time-slices are fixed. We consider a graph with 4 million nodes, 100,000 clients, and 20 dependency time-slices. It can be noted that at 1, 2, 3 and 4 dependencies, the traversal time and number of nodes traversed are small. The number of nodes traversed increases rapidly thereafter. The traversal time is approximately linear with the number of dependencies.

Summary: Our experiments show that factors that affect the performance of the Accountability Graph are number of compromised clients, number of dependencies, and the number of updates. For all of the scenarios we considered, and without performing any application-specific optimizations, the traversal time was less than 8 seconds. The number of dependencies and the number of dependencies per update seemed to be the most influential factors in increasing the traversal time.

VI. RELATED WORK

In this section we summarize related work in several research areas in distributed systems that relate to the problem we present in this paper. We note that our work is complementary to the work presented below.

a) Directed Acyclic Graphs in Operating and Distributed Systems: The core mechanism of our tool is building a causality graph. Directed acyclic graphs (DAGs) were previously used in operating systems and distributed systems. For example, the Time Warp Operating System [24] maintains two “wave fronts” of computation. The front wave front represents speculative computation that is hazarded by rollback. The rear wave front bounds the roll back. Computations on the rear wave front depend only on prior computations that are committed. Computations between the two wave fronts are tracked using causal dependency tracking (a dependency DAG), and can be canceled.

In the distributed systems field, the Trans protocol [14] and the Transis system [25] also use a DAG in order to ensure reliable delivery of multicast messages using non-reliable multicast. This DAG is maintained on the fly and updated accordingly as messages are delivered by all of the members of the group. Strom and Yemini [26] replace synchronization by causal dependency tracking in order to overcome benign process failures in distributed systems.

b) Intrusion Detection Systems: The security community has developed many mechanisms that can detect malicious users after they have penetrated a computer system. The large body of intrusion detection literature [9] testifies to our assertion that malicious intruders will sometimes gain access to even the best secured systems. We provide a way for to assess and cope with these penetrations, while intrusion detection systems provide ways to detect them. When a malicious, yet authorized, intruder is detected, systems typically alert an

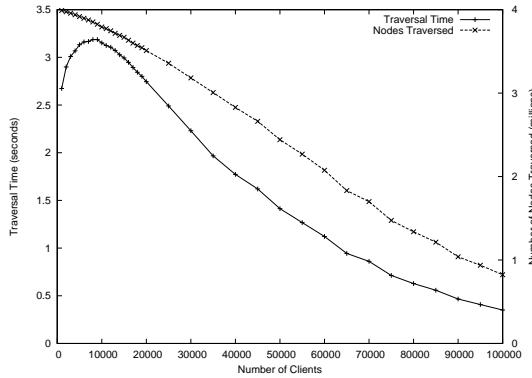


Fig. 6. Traversal time as a function of the number of clients.

administrator and may trigger an automated damage mitigation response. Traditionally, intrusion detection researchers have taken two main approaches: misuse detection and anomaly detection. Misuse detection focuses on identifying user behavior that matches a specific attack signature [7]. Anomaly detection [8] focuses on finding user behavior that deviates from normal system use. We want to emphasize that intrusion detection strategies are more than merely complementary to the Accountability Graph. Intrusion detection forms an important component of our solution since a malicious user must first be detected before the damage created by that user can be mitigated.

The BackTracker Tool by King and Chen [27] was designed to help system administrators analyze intrusions to their operating system. Working backward from a detection point such as a suspicious file or process, BackTracker identifies the events and objects that could have affected that detection point and displays chains of events in a dependency graph. The system administrator can focus the detective work on those chains of events in order to understand how the intruder gained access to the system. Our work, in comparison, assumes that the malicious clients (insiders) are authorized to access the system and simply use the application to create or inject incorrect inputs or updates. We are concerned with quickly identifying what part of the current application state is corrupt, suspected, and (no less important) not affected. The use of dependency graphs in BackTracker, together with our past directed acyclic graph work in Transis [25] to efficiently track causality, inspired our Accountability Graph approach to cope with malicious clients.

c) Byzantine-Resilient Replication: The first practical work to solve replication while withstanding Byzantine failures is the work of Castro and Liskov [1]. Their algorithm requires a number of $3f + 1$ servers in order to tolerate f faults and asks the client to wait for $f + 1$ identical answers out of $2f + 1$ answers in order to make sure that it received a correct answer. The work is fundamentally based on Byzantine Consensus, for which a good overview can be found in [28]. The Byzantine replication provides protection against malicious servers, and does not address the malicious clients problem.

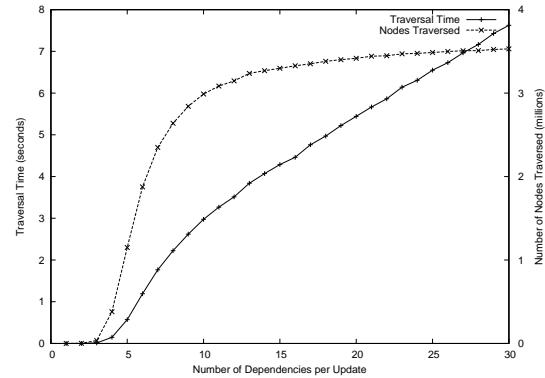


Fig. 7. Traversal time as a function of the number dependencies

VII. CONCLUSIONS

In this paper we identified a significant attack against distributed systems, mounted by malicious clients that deliberately insert incorrect data into the system using authorized channels. We proposed a generic mechanism, Accountability Graph, that tracks the dependencies between all of the updates in the system, and that can classify data as corrupt, suspect, or not affected, giving the ability to conduct risk assessment and vulnerability analysis with respect to the compromised client attack. We demonstrated the usefulness of our solution in three different applications, and we showed that the overhead associated with our solution is reasonable in these cases.

REFERENCES

- [1] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.
- [2] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault-tolerant services," in *SOSP*, 2003.
- [3] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Journal of Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.
- [4] S. Osborn, "Database security integration using role-based access control," in *IFIP WG11.3 Working Conference on Database Security*, August 2000.
- [5] S. De, C. Eastman, , and C. Farkas, "Secure access control in a multi-user database," in *ESRI User Conference*, 2002.
- [6] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *SIGMOD 2004*, 2004.
- [7] "Using decision trees to improve signature-based intrusion detection," in *RAID 2003*, 2003.
- [8] Y. Xie, H.-A. Kim, D. R. O'Hallaron, M. K. Reiter, and H. Zhang, "Seurat: A pointillist approach to anomaly detection," in *RAID*, pp. 238–257, 2004.
- [9] E. Jonsson, A. Valdes, and M. Almgren, eds., *Recent Advances in Intrusion Detection: 7th International Symposium, RAID 2004, Sophia Antipolis, France, September 15-17, 2004. Proceedings*, vol. 3224 of *Lecture Notes in Computer Science*, Springer, 2004.
- [10] C. Wilson, "Network Centric Warfare: Background and Oversight Issues for Congress; CRS Report for Congress," June 2004.
- [11] R. Wible, "Update report on alliance for building regulatory reform in the digital age," June 2004. http://www.astronaut3.com/ncsbc/content/html/update_report.htm.
- [12] *Digital Signature Standard (DSS)*. No. FIPS 186-2, National Institute for Standards and Technology (NIST), 2000. <http://csrc.nist.gov/publications/fips/fips186-2/fips186-2.pdf>.
- [13] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

- [14] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala, "Broadcast protocols for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 17–25, January 1990.
- [15] "The debian linux distribution." <http://www.debian.org/>.
- [16] "The redhat linux distribution." <http://www.redhat.com/>.
- [17] "Apache software foundation," 1999-2004. <http://www.apache.org/>.
- [18] "Gnome foundation," 2003. <http://www.gnome.org/>.
- [19] "Apache software foundation server compromised, resecured," May 2001. <http://seclists.org/lists/bugtraq/2001/May/0350.html>.
- [20] "Some debian project machines compromised," November 2003. <http://www.debian.org/News/2003/20031121>.
- [21] C. Franklin, "Gnome servers attacked putting the penguin on guard," April 2004. <http://www.networkcomputing.com/showitem.jhtml?articleID=18900826>.
- [22] "Clinicians' biodefense network," 2001. <http://www.upmc-cbn.org>.
- [23] L. Randovich, "Hopkins Center to Launch Clinicians' Biodefense Network," *Biodefense Quarterly, A publication of the Johns Hopkins Center for Civilian Biodefense Network*, vol. 4, no. 2, 2002.
- [24] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto, "Time warp operating system," in *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pp. 77–93, ACM Press, 1987.
- [25] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A communication sub-system for high availability," in *In Proceedings of the 22nd Annual International Symposium on Fault Tolerant Computing*, pp. 76–84, July 1992.
- [26] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, August 1985.
- [27] S. T. King and P. M. Chen, "Backtracking Intrusions," in *Symposium on Operationg System Principles (SOSP)*, pp. 223–236, October 2003.
- [28] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *Fundamentals of Computation Theory*, pp. 127–140, 1983.